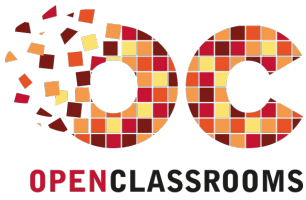
Fonctionnement d'un ordinateur depuis zéro

Par Lucas Pesenti (Lucas-84)
et Guy Grave (Mewtow)



www.openclassrooms.com

Licence Creative Commons 4 2.0
Dernière mise à jour le 6/01/2013

Sommaire

Sommaire ... 2
Lire aussi ... 7
Fonctionnement d'un ordinateur depuis zéro ... 9
Partie 1 : Tout ça rien qu'avec des 0 et des 1 ! ... 9
Un ordinateur, c'est très bête : ça ne sait pas compter jusqu'à deux ! ... 10
Nombres entiers ... 10
Différentes bases ... 10
Le binaire, la base 2 ... 11
Représentation en signe-valeur absolue ... 12
Codage en complément à 1 ... 13
Complément à deux ... 14
Nombres à virgule ... 16
Écriture scientifique ... 16
Formats de flottants ... 17
Exceptions et arrondis ... 18
Codage du texte ... 19
Standard ASCII ... 19
Unicode ... 20
Nos bits prennent la porte ! ... 20
Codage NRZ ... 21
Codage NRZ ... 21
Tensions de référence ... 21
Transistors ... 22
Transistors CMOS ... 22
Loi de Moore ... 23
Portes logiques ... 24
La porte NON ... 25
La porte ET ... 27
Porte NAND ... 28
La porte OU ... 30
Porte NOR ... 30
Porte XOR ... 32
Porte NXOR ... 33
Créons nos circuits ! ... 34
Circuits combinatoires ... 34
Tables de vérité ... 34
Équations logiques ... 35
Méthode des Minterms ... 36
Simplifications du circuit ... 38
Mais ... 39
Circuits séquentiels ... 39
Comment donner de la mémoire à nos circuits ? ... 39
Bascules ... 41
Mémoires ... 45
Tic, Tac, Tic, Tac : Le signal d'horloge ... 46
Temps de propagation ... 46
Circuits synchrones ... 47
Et dans nos PC ? ... 48
Partie 2 : Architecture de base ... 50
C'est quoi un ordinateur ? ... 50
Numérique versus analogique ... 50
Analogique versus numérique ... 50
L'immunité au bruit ... 50
Architecture de base ... 52
I/O et traitement ... 52
Automates ... 53
Programme ... 54
Ordinateurs ... 54
Organisation ... 54
Mémoire ... 55
Bus de communication ... 57
Processeur ... 58
La gestion de la mémoire ... 63
Deux mémoires pour le prix d'une ... 63
Séparation matérielle des mémoires ... 63
Architecture modifiée ... 65
L'organisation de la mémoire et la pile ... 66
Pile, Tas et Mémoire Statique ... 66
La pile ... 67
Last Input First Output ... 68
Machines à pile et successeurs ... 69
Machines à pile ... 69
Machines à accès aléatoire ... 72
Les hybrides ... 73

Partie 3 : Processeur et Assembleur ... 75
Langage machine et assembleur ... 76
Instructions ... 76
C'est quoi une instruction ? ... 76
Type des données et instructions ... 77
Longueur des données à traiter ... 78
Jeux d'instruction ... 78
RISC vs CISC ... 78
Jeux d'instructions spécialisés ... 80
Et pour nos ordinateurs ? ... 81
Registres architecturaux ... 81
A quoi servent ces registres ? ... 81
Registres architecturaux ... 83
8, 16, 32, 64 bits : une histoire de taille des registres ... 85
Représentation en binaire ... 85
Opcode ... 86
Opérandes ... 86
Modes d'adressage ... 86
Encodage du mode d'adressage ... 96
Jeux d'instructions et modes d'adressages ... 98
Longueur d'une instruction ... 98
Classes d'architectures ... 99
A accès mémoire strict ... 99
A pile ... 99
A accumulateur unique ... 99
Architectures registre-mémoire ... 100
Load-store ... 101
Un peu de programmation ! ... 101
C'est un ordre, éxecution ! ... 102
Program Counter ... 102
Les exceptions ... 103
Et que ca saute ! ... 103
Instructions de test ... 103
Branchements ... 104
Structures de contrôle, tests et boucles ... 106
Le Si...Alors ... 106
Si...Alors...Sinon ... 107
Boucles ... 108
Sous-programmes : c'est fait en quoi une fonction ? ... 110
A quoi ça sert ? ... 110
Retour vers la future (instruction) ! ... 113
Paramètres et arguments ... 113
Une histoire de registres ... 114
Valeur de retour ... 115
Variables automatiques ... 116
Plusieurs piles ... 116
Il y a quoi dans un processeur ? ... 117
Execution d'une instruction ... 118
Instruction Cycle ... 118
Micro-instructions ... 118
L'intérieur d'un processeur ... 120
Les unités de calcul ... 121
Vu de l'extérieur ... 122
A l'intérieur d'une unité de calcul ... 123
Unités annexes ... 126
Registres et interface mémoire ... 127
Registres simples ... 127
Registres non-référencables ... 128
Register File ... 128
Register Files séparés ... 131
Fichier de registre Unifié ... 131
Communication avec la mémoire ... 134
Le chemin de données ... 136
Une histoire de connexion ... 136
Chemin de donnée à un seul bus ... 137
Et avec plusieurs bus ? ... 138
Conclusion ... 142
Le séquenceur ... 142
Séquenceurs câblés ... 143
Séquenceur micro-codé ... 145
Séquenceurs hybrides ... 148
Les transport triggered architectures ... 148
L'étape de fetch ... 149
Registre pointeur instruction ... 149
Compteur ordinal ... 149
Le calcul de l'adresse suivante ... 151
Les branchements ... 152
L'exception qui confirme la règle ... 154
Les circuits d'une ALU entiére ... 154
Décalages et rotations ... 155
Décalages et rotations ... 155
Multiplexeurs ... 157

Décaleur logique ... 158
Décaleur arithmétique ... 160
Rotateur ... 161
Barell shifter ... 162
Addition ... 162
Additionneur à propagation de retenue ... 162
L'additionneur à sélection de retenue ... 164
Additionneurs à anticipation de retenue ... 165
Les Overflows ... 167
Entiers strictement positifs, non signés ... 167
Complément à deux et complément à un ... 168
Soustraction ... 170
Complément à deux et complément à un ... 170
Signe-magnitude ... 172
Comparaison ... 173
Multiplication ... 173
Entiers non-signés ... 174
Entiers signés ... 180
Array Multipliers ... 183
Tree Multipliers ... 187
Division ... 187
Division à restauration ... 187
Division sans restauration ... 190
La division SRT ... 190
Partie 4 : Mémoires ... 192
Mémoires ... 192
Des mémoires en veux-tu, en voilà ! ... 192
Capacité mémoire ... 192
Mémoires volatiles et non-volatiles ... 193
RWM ou ROM ... 194
Le temps d'accès ... 195
Mémoires RAM ... 195
Donnée, où es-tu ? ... 196
Mémoires Séquentielles ... 196
Mémoires à accès aléatoire ... 196
Mémoires FIFO ... 197
Mémoires LIFO ... 197
Content Adressables Memories ... 198
Une histoire de bus ... 199
Bus de commande ... 199
Bus d'adresse ... 200
Connexion du bus sur la mémoire ... 201
Toutes les mémoires ne se valent pas ! ... 203
Une histoire de vitesse ... 203
Registres ... 205
Cache ... 207
Local Stores ... 207
Mémoires principales ... 208
Mémoires de masse ... 208
Mémoriser un bit ... 209
Mémoire SRAM ... 209
Avec des portes logiques ... 209
Avec des transistors ... 213
Mémoire DRAM ... 215
3T-DRAM ... 215
1T-DRAM ... 218
Correction d'erreurs ... 220
Correction et détection d'erreurs ... 220
Bit de parité ou d'imparité ... 220
Mémoires ECC ... 221
Contrôleur et plan mémoire ... 221
Mémoires à adressage linéaire ... 222
Plan mémoire linéaire ... 222
Connectons le tout au bus ... 223
Décodeurs ... 224
Circuit complet ... 227
Mémoires à adressage par coïcidence ... 228
Principe ... 228
Adressage par coïncidence ... 229
Adresses hautes et basses ... 230
Mémoire à Row Buffer ... 232
Principe ... 232
Plan mémoire ... 233
Row Buffer ... 234
Sélection de colonnes ... 235
Avantages et inconvénients ... 236
Interfacage avec le bus ... 236
Circuits 3-états ... 236
Mémoires à ports de lecture et écriture séparés ... 238
Assemblages de mémoires ... 239
Arrangement horizontal ... 239
Arrangement vertical ... 241

Mémoires DDR, SDRAM et leurs cousines ... 243
Les mémoires RAM asynchrones ... 244
Format des mémoires FPM et EDO ... 244
RAS et CAS ... 245
Rafraichissement mémoire ... 248
Mémoires FPM et EDO ... 249
EDO-RAM ... 250
Les mémoires SDRAM ... 252
Pipelining des requêtes mémoires ... 252
Timings mémoires ... 253
Mode Burst ... 254
Les mémoires DDR ... 255
Principe ... 255
DDR1 ... 257
DDR2 ... 258
DDR3 ... 258
GDDR ... 259
Format DIMM et SO-DIMM ... 259
Mémoires non-volatiles ... 262
Le disque dur ... 262
C'est fait en quoi ? ... 262
Adressage d'un disque dur ... 264
Requêtes d'accès au disque dur ... 265
Mémoires FLASH ... 266
Cellule mémoire de FLASH ... 266
Mémoires FLASH MLC ... 267
Les mémoires FLASH ne sont pas des RAM ! ... 268
FLASH NAND et NOR ... 269
Les SSD ... 269
Partie 5 : Périphériques, bus, et entrées-sorties ... 269
Bus, cartes mères, chipsets et Front Side Bus ... 270
Un bus, c'est rien qu'un tas de fils... ... 270
Bus série et parallèles ... 271
Simplex, Half duplex ou Full duplex ... 273
Bus synchrones et asynchrones ... 273
Va falloir partager ! ... 275
Conflit d'accès ... 276
Arbitrage par multiplexage temporel ... 276
Arbitrage par requête ... 276
Chipset, back-plane bus, et autres ... 276
Première génération ... 277
Seconde génération ... 277
De nos jours ... 278
Architectures sans Front side bus ... 280
Communication avec les Entrées-Sorties ... 281
Interfacage Entrées-sorties ... 282
Interfaçage ... 282
Registres d'interfaçage ... 282
Contrôleur de périphérique ... 283
Problèmes ... 285
Interruptions ... 286
Déroulement d'une interruption ... 286
Les différents types d'interruptions ... 287
Direct Memory Access ... 289
Arbitrage du bus ... 289
Direct Memory Acces ... 289
Adressage des périphériques ... 293
Connexion directe ... 293
Bus multiples ... 293
Bus d'entrées-sorties multiplexé ... 293
Espace d'adressage séparé ... 294
Partage d'adresse ... 295
IO Instructions ... 296
Entrées-sorties mappées en mémoire ... 296
Memory Mapped I/O ... 296
Bus unique ... 298
Et pour le CPU ? ... 299
Partie 6 : Hiérarchie mémoire ... 300
La mémoire virtuelle ... 300
Solutions matérielles ... 300
Mémoire virtuelle ... 300
La MMU ... 301
Segmentation ... 301
Principe ... 302
Relocation ... 303
Protection mémoire ... 306
Allocation dynamique ... 309
Partage de segments ... 310
Pagination ... 311
Swapping ... 312
Remplacement des pages mémoires ... 312

Translation d'adresse ... 313
Allocation dynamique ... 315
Protection mémoire ... 315
Les mémoires caches ... 316
Accés au cache ... 316
Accès au cache ... 316
Écriture dans un cache ... 317
Cache bloquant et non-bloquant ... 319
Localité spatiale et temporelle ... 319
Localité temporelle ... 319
Localité spatiale ... 320
L'influence du programmeur ... 320
Correspondance Index - Adresse ... 321
Tag d'une ligne de cache ... 321
Adresses physiques ou logiques ? ... 321
Les caches direct mapped ... 323
Les caches Fully associative ... 326
Les caches Set associative ... 327
Remplacement des lignes de cache ... 329
Remplacement des lignes de cache ... 330
Aléatoire ... 330
FIFO : First Input First Output ... 331
MRU : Most Recently Used ... 332
LFU : Last Fréquently Used ... 333
LRU : Last Recently Used ... 334
Approximations du LRU ... 334
LRU amélioré ... 335
On n'a pas qu'un seul cache ! ... 335
Caches L1, L2 et L3 ... 335
Caches d'instruction ... 337
Caches spécialisés ... 338
Le Prefetching ... 338
Array Prefetching ... 339
Prefetchers séquentiels ... 339
History based prefecther ... 342
Le Futur ... 343
Linked Data Structures Prefetching ... 343
Linked Data Structures ... 343
Dependance Based Prefetching ... 345
Runahead Data Prefetching ... 347
Instruction Prefetching ... 347
Prefetching séquentiel, le retour ! ... 347
Target Line Prefetching ... 348
Wrong Path Prediction ... 349
Autres ... 350
Partie 7 : Le parallélisme d'instruction et les processeurs modernes ... 351
Le pipeline : qu'est-ce que c'est ? ... 351
Un besoin : le parallèlisme d'instruction ... 351
Le pipeline : rien à voir avec un quelconque tuyau à pétrole ! ... 352
Sans pipeline ... 353
Et dieu inventa le pipeline ... 353
Etages, circuits et fréquence ... 354
Un besoin : isoler les étages du pipeline ... 354
Comment on fait ? ... 356
Une histoire de fréquence... ... 357
Implémentation hardware ... 359
Pipeline à 7 étages ... 360
Datapath ... 360
Signaux de commandes ... 362
Branchements ... 363
Pipelines complexes ... 366
Micro-opérations ... 366
Instructions multicyles ... 370
Interruptions et Pipeline ... 372
NOP Insertion ... 373
In-Order Completion ... 374
Ordre des écritures ... 374
Result Shift Register ... 375
Speculation Recovery ... 376
Out Of Order Completion ... 377
Register Checkpointing ... 377
Re-Order Buffer ... 377
History Buffer ... 378
Future File ... 380
Les branchements viennent mettre un peu d'ambiance ! ... 381
Solutions non-spéculatives ... 382
Délai de branchements ... 382
Branch Free Code ... 383
Instructions à prédicats ... 383
Les processeurs malins ... 384
Conclusion ... 384
Prédiction de branchement ... 384

Erreurs de prédiction ... 385
Prédiction de direction de branchement ... 388
Prédiction de branchement ... 390
Eager execution ... 398
Quelques limites pratiques ... 398
Disjoint Eager Execution ... 400
Dépendances de données ... 402
Dépendances d'instructions ... 402
Dépendances structurelles ... 402
Dépendances de données ... 403
Que faire ? ... 405
Pipeline Bubble / Stall ... 405
Principe ... 405
Processeurs In-order ... 407
Implémentation ... 408
Bypass et Forwarding ... 408
Effet des dépendances RAW ... 409
Bypass ... 409
Implémentation ... 410
Clusters ... 412
Execution Out Of Order ... 413
Principe ... 413
Une idée géniale ... 413
Deux types d'Out Of Order ... 414
Scoreboarding ... 414
Pipeline d'un processeur Scoreboardé ... 415
Scoreboard ... 416
Final ... 417
Out Of Order Issue ... 417
Centralized Instruction Window ... 417
Instructions Windows Multiples ... 418
Quelques détails ... 419
L'algorithme de Tomasulo et le renommage de registres ... 420
Le renommage de registres ... 421
Des dépendances fictives ... 421
Le renommage de registres ... 421
Des registres en double ! ... 422
C'est fait comment ? ... 422
Reservations stations ... 422
Aperçu ... 422
Issue ... 423
Reservation Stations ... 423
Dispatch ... 425
Le Common Memory Bus ... 426
Accès mémoires ... 427
Bilan ... 427
Re-Orders Buffers ... 427
Le Reorder Buffer ... 428
Une File ... 429
Spéculation Recovery ... 431
Accès mémoire ... 432
Autres formes de renommages ... 432
ROB ... 432
Rename Register File ... 433
Physical Register File ... 434
L'unité de renommage ... 435
Register Map Table ... 435
Implémentation ... 436
Les optimisations des accès mémoire ... 437
Dépendances, le retour ! ... 438
Utilité ... 438
De nouvelles dépendances ... 438
Dépendances de nommage ... 439
Store queue ... 439
Bypass Store Queue ... 439
Dependances d'alias ... 439
Vérifications des adresses ... 440
Exécution spéculative ... 440
Memory Dependance Prediction ... 441
Load Adress Prediction ... 442
Last Adress ... 442
Stride ... 442
Context Based Predictor ... 443
Efficacité ... 444
Load Value Prediction ... 444
Value prediction ... 444
Pourquoi ça marche ? ... 444
Implémentation ... 445
Efficacité ... 445
Processeurs Multiple Issue ... 446
Processeurs superscalaires ... 447
Processeurs superscalaires ... 447

Superscalaire In Order versus Out Of Order ... 448
Fetch ... 449
Décodeur d'instruction ... 449
Influence sur l'unité de renommage ... 450
Processeurs VLIW ... 452
Bundles ... 452
Problèmes ... 453
Processeurs EPIC ... 454
Bundles ... 454
Prédication ... 454
Delayed Exceptions ... 455
Spéculation sur les lectures ... 455
Large Architectural Register File ... 458
Bilan ... 458
Partie 8 : Annexes ... 458
Alignement mémoire et endianess ... 459
Alignement mémoire ... 459
Accès mémoires à la granularité de l'octet ... 459
Alignement supérieur à l'octet ... 461
Endianness ... 464
Big Endian ... 464
Little Endian ... 464
Bi-Endian ... 465
Liens sur le Siteduzéro ... 465
Remerciements ... 466

Fonctionnement d'un ordinateur depuis zéro

Par

Guy Grave (Mewtow) et Lucas Pesenti (Lucas-84)

Mise à jour : 06/01/2013
Difficulté : Facile Durée d'étude : 1 mois, 3 jours, 3 heures, 7 minutes
(cc) BY-NC-ND

Vous vous êtes déjà demandé comment fonctionne un ordinateur ou ce qu'il y a dedans ?

Alors ce tutoriel est fait pour vous.

Dans ce cours, vous allez apprendre ce qu'il y a dans notre ordinateur, ce qui se passe à l'intérieur de votre processeur ou de votre mémoire RAM. Vous saurez tout des dernières innovations présentes dans nos processeurs, pourquoi la course à la fréquence est terminée, ou encore comment fabriquer des registres. On commencera par des choses simples comme le binaire, pour arriver progressivement jusqu'au fonctionnement des derniers processeurs, en passant par plein de choses passionnantes comme l'assembleur, les mémoires caches, et d'autres choses encore !

Ce tutoriel ne posera pas de soucis, même pour ceux qui n'ont jamais programmé ou qui débutent tout juste : ce cours est accessible à n'importe qui, sans vraiment de prérequis. En clair : on part de zéro !

Partie 1 : Tout ça rien qu'avec des 0 et des 1 !

Quitte à parler d'architecture des ordinateurs, autant commencer par les bases, non ? On va donc voir comment l'ordinateur va se représenter les informations (sons, images, vidéos...) en utilisant le binaire. On va aussi voir en quoi il est fait, et quels sont les circuits de base qui le compose. Vous y apprendrez aussi à concevoir des circuits assez simples.

Un ordinateur, c'est très bête : ça ne sait pas compter jusqu'à deux !

On a sûrement déjà dû vous dire qu'un ordinateur comptait uniquement avec des zéros et des uns. Et bien sachez que c'est vrai : on dit que notre ordinateur utilise la numération binaire.

Le binaire, qu'est-ce que c'est que ce truc ?

C'est juste une façon de représenter un nombre en utilisant seulement des 0 et des 1. Et un ordinateur ne sait compter qu'en binaire. Toutefois, le binaire ne sert pas qu'à stocker des nombres dans notre ordinateur. Après tout, votre ordinateur ne fait pas que manipuler des nombres : il peut aussi manipuler du texte, de la vidéo, du son, et pleins d'autres choses encore. Eh bien, sachez que tout cela est stocké... avec uniquement des 0 et des 1. Que ce soit du son, de la vidéo, ou tout autre type de donnée manipulable par notre ordinateur, ces données sont stockées sous la forme de suites de zéros et de uns que notre ordinateur pourra manipuler comme bon lui semble.

Pour comprendre le fonctionnement d'un ordinateur, on va donc devoir aborder le binaire. Nous allons commencer par voir comment sont stockées quelques données de base comme les nombres ou le texte. Et pour cela, nous allons commencer par un petit rappel pour ceux qui n'ont jamais été en CM1.

Nombres entiers

Nous allons commencer par parler des nombres entiers.

Dans notre système de représentation décimal, nous utilisons dix chiffres pour écrire nos nombres entiers positifs : ***0***, ***1***, ***2***, ***3***, ***4***, ***5***, ***6***, ***7***, ***8*** et ***9***.

Prenons le nombre 1337. Le chiffre le plus à droite est le chiffre des **unités**, celui à côté est pour les **dizaines**, suivi du chiffre des **centaines**...
Cela nous donne :

$$1 \times 1000 + 3 \times 100 + 3 \times 10 + 7 \times 1$$

Jusque là vous devez vous ennuyer, non (Enfin j'espère !) ?
Bref, reprenons notre nombre 1337. On va remplacer les unités, dizaines, centaines et milliers par leurs puissances de dix respectives :

$$1 \times 10^3 + 3 \times 10^2 + 3 \times 10^1 + 7 \times 10^0$$

Tous les nombres entiers qui existent peuvent eux aussi être écrits sous cette forme : on peut les décomposer en une somme de multiples de puissances de 10. Lorsque c'est le cas, on dit qu'ils sont en **base 10**.

Différentes bases

Ce qui peut être fait avec des puissances de 10 peut être fait avec des puissances de 2, 3, 4, 125, etc : on peut utiliser d'autres bases que la base 10. Rien n'empêche de décomposer un nombre en une somme de multiples de puissance de 2, ou de 3, par exemple. On peut ainsi utiliser d'autres bases.

En informatique, on utilise rarement la base 10 à laquelle nous sommes tant habitués. Nous utilisons à la place deux autres bases :

- La base 2 (système **binaire**) : les chiffres utilisés sont 0 et 1 ;
- La base 16 (système **hexadécimal**) : les chiffres utilisés sont 0, 1, 2, 3, 4, 5, 6, 7, 8 et 9 ; auxquels s'ajoutent les six premières lettres de notre alphabet : A, B, C, D, E et F.

Voici le tableau des 16 premiers nombres des bases citées ci-dessus :

Base 10	Base 2	Base 16
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Le binaire, la base 2

Le binaire, c'est la base 2. Seuls deux chiffres sont utilisés : 0 et 1. Lorsque vous écrivez un nombre en binaire, celui-ci peut toujours être écrit sous la forme d'une somme de puissances de 2.

Par exemple 6 s'écrira donc 0110 en binaire : $0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 6$

En général, tout nombre en binaire s'écrit sous la forme
$a_0 \times 2^0 + a_1 \times 2^1 + a_2 \times 2^2 + a_3 \times 2^3 + a_4 \times 2^4 + ... + a_n \times 2^n$.

Les coefficients $a_0, a_1, a_2...$ valent 1 ou 0. Ces coefficients ne sont rien d'autres que les "chiffres" de notre nombre écrit en base 2. Ces "chiffres" d'un nombre codé en binaire sont aussi appelés des **bits**. Pour simplifier, on peut dire qu'un bit est un truc qui vaut 0 ou 1.

L'exposant qui correspond à un bit a_n est appelé le **poids** du bit. Le bit de poids faible est celui qui a la plus petite valeur dans un nombre : c'est celui qui est le plus à droite du nombre (si vous écrivez vos nombres dans le bon sens, évidemment). Le bit de poids fort c'est l'inverse, évidemment : c'est celui qui est placé le plus à gauche.

Capacité

Petite remarque assez importante : avec n bits, on peut coder 2^n valeurs différentes, dont le 0. Ce qui fait qu'on peut compter de 0 à $2^n - 1$. N'oubliez pas cette petite remarque : elle sera assez utile dans le suite de ce tutoriel.

Changement de base

La représentation des entiers positifs en binaire est très simple : il suffit simplement de changer de base, et de passer de la base 10 à la base 2. Il existe un algorithme qui permet de changer un nombre en base décimale vers un nombre en base binaire : il

consiste à diviser itérativement le quotient de la division précédente par 2, et de noter le reste. Enfin, il faut lire de bas en haut les restes trouvés.

Exemple :

34/2 = 17 reste : 0
17/2 = 8 reste : 1
8/2 = 4 reste : 0
4/2 = 2 reste : 0
2/2 = 1 reste : 0
1/2 = 0 reste : 1

Soit 100010 en binaire.

Représentation en signe-valeur absolue

Bref, maintenant qu'on a vu les entiers strictement positifs ou nuls, on va voir comment faire pour représenter les entiers négatifs n binaire. Avec nos 1 et nos 0, comment va-t-on faire pour représenter le signe moins ("-") ? Eh bien, il existe plusieurs méthodes. Les plus utilisées sont :

- La représentation en **signe-valeur absolue** ;
- La représentation en **complément à un** ;
- La représentation en **complément à deux**.

La solution la plus simple pour représenter un entier négatif consiste à coder sa valeur absolue en binaire, et rajouter un **bit de signe** au tout début du nombre. Ce bit servira à préciser si c'est un entier positif ou un entier négatif. C'est un peu la même chose qu'avec les nombres usuels : pour écrire un nombre négatif, on écrit sa valeur absolue, en plaçant un moins devant. Ici, c'est la même chose, le bit de signe servant de signe moins (quand il vaut 1) ou plus (quand il vaut 0).

Bit de signe	Nombre codé en binaire sur n bits

Par convention, ce bit de signe est égal à :

- **0** si le nombre est **positif** ;
- **1** si le nombre est **négatif**.

Exemple :

- Codage de 34 sur 8 bits :
  34 = 0010 0010
- Codage de -34 sur 8 bits :
  -34 = 1010 0010

Capacité

En utilisant n bits, bit de signe inclut, un nombre codé en représentation signe-valeur absolue peut prendre toute valeur comprise entre $-(\frac{2^n}{2}-1)$ et $\frac{2^n}{2}-1$. Cela vient du fait qu'on utilise un bit pour le signe : il reste alors N-1 bits pour coder les valeurs absolues. Ces N-1 bits permettent alors de coder des valeurs absolues allant de 0 à $\frac{2^n}{2}-1$.

Avec 4 bits, cela donne ceci :

	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Unsigned	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Sign & Magnitude	0	1	2	3	4	5	6	7	-0	-1	-2	-3	-4	-5	-6	-7

On remarque que l'intervalle des entiers représentables sur N bits est symétrique : pour chaque nombre représentable sur n bits en représentation signe-valeur absolue, son inverse l'est aussi.

Désavantages

Vous avez certainement remarqué que le zéro, est représentable par deux entiers signés différents, quand on utilise la représentation signe-magnitude.
Exemple avec un nombre dont la valeur absolue est codée sur 8 bits, et un bit de signe au début. Le bit de signe est coloré en rouge.
0 0000 0000 = 0
1 0000 0000 = -0, ce qui est égal à zéro.

Comme vous le voyez sur cet exemple, le zéro est présent deux fois : un -0, et un $+0$. Cela peut parfois poser certains problèmes, lorsqu'on demande à notre ordinateur d'effectuer des calculs ou des comparaisons avec zéro par exemple.

Il y a un autre petit problème avec ces entiers signe-valeur absolue : faire des calculs dessus est assez compliqué. Comme on le verra plus tard, nos ordinateurs disposent de circuits capables d'additionner, de multiplier, diviser, ou soustraire deux nombres entiers. Et les circuits capables de faire des opérations sur des entiers représentés en signe-magnitude sont compliqués à fabriquer et assez lents, ce qui est une désavantage.

Codage en complément à 1

Passons maintenant à une autre méthode de codage des nombres entiers qu'on appelle le **codage en complément à 1**. Cette méthode est très simple. Si le nombre à écrire en binaire est positif, on le convertit en binaire, sans rien faire de spécial. Par contre, si ce nombre est un nombre négatif, on code sa valeur absolue en binaire et on inverse tous les bits du nombre obtenu : les 0 deviennent des 1, et vice-versa.

Avec cette méthode, on peut remarquer que le bit de poids fort (le bit le plus à gauche) vaut 1 si le nombre est négatif, et 0 si le nombre représenté est positif. Celui-ci se comporte comme un bit de signe. Par contre, il y a un petit changement comparé à la représentation en signe-valeur absolue : le reste du nombre (sans le bit de signe) n'est pas égal à sa valeur absolue si le nombre est négatif.

Capacité

En utilisant n bits, un nombre représenté en complément à un peut prendre toute valeur comprise entre $-(\frac{2^n}{2}-1)$ et $\frac{2^n}{2}-1$: pas de changements avec la représentation signe-valeur absolue.

Par contre, les nombres ne sont pas répartis de la même façon dans cet intervalle. Regardez ce que ça donne avec 4 bits pour vous en convaincre :

	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Unsigned	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1's Compl.	0	1	2	3	4	5	6	7	-7	-6	-5	-4	-3	-2	-1	-0

Désavantages

Cette méthode est relativement simple, mais pose exactement les mêmes problèmes que la représentation signe-magnitude. Le zéro est toujours représenté par deux nombres différents : un nombre ne contenant que des 0 (0000 0000 ...), et un nombre ne contenant que des 1 (1111 1111 ...). Pour la complexité des circuits, la situation est un peu meilleure qu'avec la représentation en signe-valeur absolue. Mais les circuits manipulant des nombres en complément à un doivent gérer correctement la présence de deux zéros, ce qui ajoute un peu de complexité inutilement. Il faut avouer que ces problèmes méritent bien une solution !

Pour faciliter la vie des concepteurs de circuits ou des programmeurs, on préfère utiliser une autre représentation des nombres entiers, différente du complément à 1 et de la représentation signe-valeur absolue, qui permet de faire des calculs simplement, sans avoir à utiliser de circuits complexes, et avec laquelle le zéro ne pose pas de problèmes.

Complément à deux

Pour éviter ces problèmes avec le zéro et les opérations arithmétiques, on a dû recourir à une astuce : on ne va utiliser que des entiers non-signés et se débrouiller avec çà. L'idée derrière la méthode qui va suivre est de coder un nombre entier négatif par un nombre positif non-signé en binaire, de façon à ce que les résultats des calculs effectués avec ce nombre positif non-signé soient identiques avec ceux qui auraient étés faits avec notre nombre négatif. Par contre, pour les nombres positifs, rien ne change au niveau de leur représentation en binaire.

Pour cela, on va utiliser les règles de l'arithmétique modulaire. Si vous ne savez pas ce que c'est, ce n'est pas grave ! Il vous faudra juste admettre une chose : nos calculs seront faits sur des entiers ayant un nombre de bits fixé une fois pour toute. En clair, si un résultat dépasse ce nombre de bits fixé (qu'on notera N), on ne gardera que les N bits de poids faible (les N bits les plus à droite).

Prenons un exemple : prenons des nombres entiers non-signés de 4 bits. Ceux-ci peuvent donc prendre toutes les valeurs entre 0 et 15. Prenons par exemple **13** et **3**. **13 + 3 = 16**, comme vous le savez. Maintenant, regardons ce que donne cette opération en binaire.

$$1101 + 0011 = 10000.$$

Ce résultat dépasse 4, qui est le nombre de bits fixé. On doit donc garder uniquement les 4 bits de poids faible et on va virer les autres.
Et voici le résultat :

$$1101 + 0011 = 0000.$$

En clair, avec ce genre d'arithmétique, $13 + 3 = 0$! On peut aussi reformuler en disant que $13 = -3$, ou encore que $3 = -13$.

Et ne croyez pas que ça marche uniquement dans cet exemple : cela se généralise assez rapidement. Pire : ce qui marche pour l'addition marche aussi pour les autres opérations, tel la soustraction ou la multiplication. Un nombre négatif va donc être représenté par un entier positif strictement équivalent dans nos calculs qu'on appelle son **complément à deux**.

Capacité

En utilisant n bits, un nombre représenté en complément à deux peut prendre toute valeur comprise entre $-\frac{2^n}{2}$ et $\frac{2^n}{2} - 1$: cette fois, l'intervalle n'est pas symétrique. Au passage, avec la méthode du complément à deux, le zéro n'est codé que par un seul nombre binaire.

Exemple avec des nombres codés sur 4 bits

	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Unsigned	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2's Compl.	0	1	2	3	4	5	6	7	-8	-7	-6	-5	-4	-3	-2	-1

Au fait : je ne sais pas si vous avez remarqué, mais le bit de poids fort (le bit le plus à gauche) vaut 1 si le nombre est négatif, et 0 si le nombre représenté est positif. Celui-ci se comporte comme un bit de signe.

Conversion entier -> binaire

Ça a l'air joli, mais comment je fais pour trouver quel est l'entier positif qui correspond à -15, ou à -50 ? Il faut bien que çà serve ton truc, non ?

Ce complément à deux se calcule en plusieurs étapes :

- 1 - On convertit notre nombre en complément à un, en inversant tous les bits du nombre.
- 2 - On ajoute 1 au résultat : on obtient alors le complément à deux de notre nombre. Ce complément à deux est alors strictement équivalent au nombre d'origine, du point de vue de l'addition, de la multiplication, de la soustraction, etc.

Pas convaincu ? alors on va prendre un exemple : 7 + (-6). On suppose que ces nombres sont codés sur quatre bits.
Pour 7, pas de changements, ça reste 0111. Pour coder -6, on va :

- prendre 6 : 0110 ;
- calculer son complément à 1 : 1001 ;
- calculer son complément à 2 : 1010.

Ensuite, il nous faut faire l'addition : 0111 + 1010 = 10001.

Et là, on prend en compte le fait que nos deux nombres de base sont codés sur 4 bits ! On ne doit garder que les 4 derniers bits de notre résultat. Le résultat de 0111 + 1010 = 10001, une fois tronqué sur 4 bits, donnera alors 0001. On trouve bien le bon résultat.

Sign Extend

Dans nos ordinateurs, tous les nombres sont représentés sur un nombre fixé et constant de bits. Ainsi, les circuits d'un ordinateur ne peuvent manipuler que des nombres de 4, 8, 12, 16, 32, 48, 64 bits, suivant l'ordinateur. Si l'on veut utiliser un entier codé sur 16 bits et que l'ordinateur ne peut manipuler que des nombres de 32 bits, il faut bien trouver un moyen de convertir notre nombre de 16 bits en un nombre de 32 bits, sans changer sa valeur et en conservant son signe. Cette conversion d'un entier en un entier plus grand, qui conserve valeur et signe s'appelle l'**extension de signe**, ou *sign extend*.

L'extension de signe des nombres positif ne pose aucun problème : il suffit de remplir les bits à gauche de notre nombre de base avec des 0 jusqu'à arriver à la taille voulue. C'est la même chose qu'en décimal : rajouter des zéros à gauche d'un nombre ne changera pas sa valeur. Cela marche quelque soit la représentation utilisée, que ce soit la représentation signe-valeur absolue, le complément à 1 ou complément à 2.

Exemple, si je veux convertir l'entier positif **0100 0101**, prenant 8 bits, en l'entier équivalent mais utilisant 16 bits, il me suffit de remplir les 8 bits à gauche de **0100 0101** par des **0**. On obtient ainsi **0000 0000 0100 0101**.

Pour les nombres négatifs, la conversion dépend de la représentation utilisée. Avec le complément à 2, l'extension de signe d'un entier négatif est simple à effectuer : il suffit de remplir les bits à gauche du nombre à convertir avec des 1, jusqu'à obtenir le bon nombre de bits.

Exemple, prenons le nombre -128, codé sur 8 bits en complément à deux : **1000 0000**. On veut le convertir en nombre sur 16 bits. Il suffit pour cela de remplir les 8 bits de poids fort (les 8bits les plus à gauche) de **1** : on obtient **1111 1111 1000 000**.

L'extension de signe d'un nombre codé en complément à 2 se résume donc en une phrase.

> Pour un nombre codé en complément à deux, il suffit de recopier le bit de poids fort de notre nombre à convertir à gauche de celui-ci jusqu'à atteindre le nombre de bits voulu.

Nombres à virgule

On sait donc comment sont stockés nos nombres entiers dans un ordinateur. Néanmoins, les nombres entiers ne sont pas les seuls nombres que l'on utilise au quotidien : il nous arrive d'utiliser des nombres à virgule. Notre ordinateur n'est pas en reste : il est lui aussi capable de manipuler des nombres à virgule sans trop de problèmes (même si de ce point de vue, certains ordinateurs se débrouillent mieux que d'autres). Notre ordinateur va parfaitement pouvoir manipuler des nombres virgule.

Il existe deux méthodes pour coder des nombres à virgule en binaire :

- La **virgule fixe** ;
- La **virgule flottante**.

La méthode de la virgule fixe consiste à émuler nos nombres à virgule à partir de nombre entiers. Un nombre à virgule fixe est donc codé par un nombre entier proportionnel à notre nombre à virgule fixe. Pour obtenir la valeur de notre nombre à virgule fixe, il suffit de diviser l'entier servant à le représenter par un nombre constant, fixé une bonne fois pour toute.

Par exemple, pour coder 1,23 en virgule fixe, on peut choisir comme "facteur de conversion" 1000. L'entier permettant de coder 1,23 sera alors 1230. La représentation en virgule fixe était utile du temps où les ordinateurs n'intégraient pas de circuits capables de travailler directement sur des nombres à virgule flottante. Cette méthode n'est presque plus utilisée, et vous pouvez l'oublier sans problème.

Les nombres à virgule fixe ont aujourd'hui étés remplacés par les nombres à virgule flottante. Ce sont des nombres dont le nombre de chiffre après la virgule est variable. De nombreuses méthodes existent pour représenter ces nombres à virgule qui sont souvent incompatibles entre-elles.

Les concepteurs de matériel électronique se sont dit qu'il fallait normaliser le stockage des flottants en mémoire ainsi que les résultats des calculs afin que tous les ordinateurs supportent les mêmes flottants et pour que les calculs flottants donnent les mêmes résultats quelque soit l'ordinateur. C'est ainsi qu'est née la **norme IEEE754**.

Cette norme IEEE754 impose diverses choses concernant nos flottants. Elle impose une façon d'organiser les bits de nos nombres flottants en mémoire, standardisée par la norme. Il faut tout de même noter qu'il existe d'autres normes de nombres flottants, moins utilisées.

Écriture scientifique

L'écriture d'un nombre flottant en binaire est basée sur son écriture scientifique. Cela permet de coder beaucoup plus de valeurs qu'un nombre en virgule fixe, à nombre de bits égal. Pour rappel, en décimal, l'écriture scientifique d'un nombre consiste à écrire celui-ci comme un produit entre un nombre et une puissance de 10. Ainsi, un nombre x aura une écriture scientifique en base 10 de la forme :

$$a \times 10^{Exposant}$$

Notre nombre a ne possède qu'un seul chiffre à gauche de la virgule : on peut toujours trouver un exposant tel que ce soit le cas. En clair, en base 10, sa valeur est comprise entre 1 (inclus) et 10 (exclu).

En binaire, c'est à peu près la même chose, mais avec une puissance de deux. L'écriture scientifique binaire d'un nombre consiste à écrire celui-ci sous la forme

$$a \times 2^{exposant}$$

Le nombre a ne possède toujours qu'un seul chiffre à gauche de la virgule, comme en base 10. Le seul truc, c'est qu'en binaire, seuls deux chiffres sont possibles : 0 et 1. Le chiffre de a situé à gauche de la virgule est donc soit un zéro ou un 1.

Pour stocker cette écriture scientifique avec des zéros et des un, il nous faut stocker la partie fractionnaire de notre nombre a, qu'on appelle la **mantisse** et l'**exposant**. On rajoute souvent un **bit de signe** qui sert à calculer le signe du nombre flottant : ce bit vaut 1 si ce nombre est négatif et vaut 0 si notre flottant est positif.

Bit de signe	Exposant	Mantisse
0	0011 0001	111 0000 1101 1001

Mantisse

Mais parlons un peu de cette mantisse. Vous croyez surement que l'ensemble de cette mantisse est stockée dans notre nombre flottant. Et bien rien n'est plus faux : seule la partie fractionnaire est stockée dans nos nombres flottants : le chiffre situé à gauche de la virgule n'est pas stocké dans la mantisse. Ce bit est stocké dans notre nombre flottant de façon implicite et peut se déduire en fonction de l'exposant : on ne doit pas le stocker dans notre nombre flottant, ce qui permet d'économiser un bit. Il est souvent appelé le **bit implicite** dans certains livres ou certaines documentations. Dans la majorité des cas, il vaut 1, et ne vaut 0 que dans quelques rares exceptions : les flottants dénormaux. On verra ceux-ci plus tard.

Exposant

Après avoir stocké notre mantisse, parlons de l'exposant. Sachez que celui-ci peut être aussi bien positif que négatif : c'est pour permettre de coder des nombres très petits. Mais notre exposant n'est pas codé avec les représentations de nombres entiers qu'on a vues au-dessus. A la place, notre exposant est stocké en lui soustrayant un décalage prédéterminé. Pour un nombre flottant de n bits, ce décalage vaut $2^{n-1} - 1$.

Formats de flottants

La norme IEEE754 impose diverses choses concernant la façon dont on gère nos flottants. Elle impose un certain format en mémoire : les flottants doivent être stockés dans la mémoire d'une certaine façon, standardisée par la norme. Elle impose une façon d'organiser les bits de nos nombres flottants en mémoire. Cette norme va (entre autres) définir quatre types de flottants différents. Chacun de ces types de flottants pourra stocker plus ou moins de valeurs différentes.
Voici ces types de flottants :

Format	Nombre de bits utilisés pour coder un flottant	Nombre de bits de l'exposant	Nombre de bits pour la mantisse
Simple précision	32	8	23
Simple précision étendue	Au moins 43	Variable	Variable
Double précision	64	11	52
Double précision étendue	80 ou plus	15 ou plus	64 ou plus

IEEE754 impose aussi le support de certains nombres flottants spéciaux. Parmi eux, on trouve **l'infini** (aussi bien en négatif qu'en positif), la valeur **NaN**, utilisée pour signaler des erreurs ou des calculs n'ayant pas de sens mathématiquement, ou des nombres spéciaux nommés les **dénormaux** qui représentent des valeurs très petites et qui sont utilisés dans des scénarios de calcul assez particuliers.

Flottants dénormalisés

Commençons notre revue des flottants spéciaux par les dénormaux, aussi appelés flottants dénormalisés. Pour ces flottants, l'exposant prend la plus petite valeur possible. Ces flottants ont une particularité : le bit implicite attaché à leur mantisse vaut 0.

Bit de signe	Exposant	Mantisse
1 ou 0	Le plus petit exposant possible	Mantisse différente de zéro

Le zéro

Le zéro est un flottant dénormalisé spécial. Sa seule particularité est que sa mantisse est nulle.

Bit de signe	Exposant	Mantisse

1 ou 0	Le plus petit exposant possible	0

Au fait, remarquez que le zéro est codé deux fois à cause du bit de signe. Si vous mettez l'exposant et la mantisse à la bonne valeur de façon à avoir zéro, le bit de signe pourra valoir aussi bien 1 que 0 : on se retrouve avec un -0 et un $+0$.

Amusons-nous avec l'infini !

Plus haut, j'ai dit que les calculs sur les flottants pouvaient poser quelques problèmes. Essayez de calculer $\frac{5}{0}$ par exemple. Si vous vous dites que votre ordinateur ne pourra pas faire ce calcul, c'est raté cher lecteur ! Le résultat sera un flottant spécial qui vaut $+\infty$. Passons sous le tapis la rigueur mathématique de ce résultat, c'est comme ça.

$+\infty$ est codé de la façon suivante :

Bit de signe	Exposant	Mantisse
0	Valeur maximale possible de l'exposant	0

Il faut savoir qu'il existe aussi un flottant qui vaut $-\infty$. Celui-ci est identique au flottant codant $+\infty$ à part son bit de signe qui est égal à 1.

Bit de signe	Exposant	Mantisse
1	Valeur maximale possible de l'exposant	0

Et le pire, c'est qu'on peut effectuer des calculs sur ces flottants infinis. Mais cela a peu d'utilité.
On peut donner comme exemple :

- L' addition ou soustraction d'un nombre réel fini à un de ces deux infinis, qui ne changera rien à l'infini de départ.
- Idem pour la multiplication par un nombre positif : $5 \times \infty$ aura pour résultat ∞.
- La multiplication par un nombre négatif changera le signe de l'infini. Par exemple, $-5 \times \infty$ aura pour résultat $-\infty$.

NaN

Mais malheureusement, l'invention des flottants infinis n'a pas réglé tous les problèmes. On se retrouve encore une fois avec des problèmes de calculs avec ces infinis.
Par exemple, quel est le résultat de $\infty - \infty$? Et pour $\frac{\infty}{-\infty}$? Ou encore $\frac{0}{0}$?

Autant prévenir tout de suite : mathématiquement, on ne peut pas savoir quel est le résultat de ces opérations. Pour pouvoir résoudre ces calculs dans notre ordinateur sans lui faire prendre feu, il a fallu inventer un nombre flottant qui signifie "je ne sais pas quel est le résultat de ton calcul pourri". Ce nombre, c'est NAN.
Voici comment celui-ci est codé :

Bit de signe	Exposant	Mantisse
1 ou 0, c'est au choix	Valeur maximale possible de l'exposant	Différent de zéro

NAN est l'abréviation de Not A Number, ce qui signifie : n'est pas un nombre. Pour être plus précis, il existe différents types de NaN, qui diffèrent par la valeur de leur mantisse, ainsi que par les effets qu'ils peuvent avoir. Malgré son nom explicite, on peut faire des opérations avec NAN, mais cela ne sert pas vraiment à grand chose : une opération arithmétique appliquée avec un NAN aura un résultat toujours égal à NAN.

Exceptions et arrondis

La norme impose aussi une gestion de certains cas particuliers. Ces cas particuliers correspondant à des erreurs, auxquelles il faut bien "répondre". Cette réponse peut être un arrêt de l'exécution du programme fautif, ou un traitement particulier (un arrondi par exemple).
En voici la liste :

- ***Invalid operation*** : opération qui produit un NAN.
- ***Overflow*** : résultat trop grand pour être stocké dans un flottant. Le plus souvent, on traite l'erreur en arrondissant le résultat vers $+\infty$.
- ***Underflow*** : pareil, mais avec un résultat trop petit. Le plus souvent, on traite l'erreur en arrondissant le résultat vers 0.
- **Division par zéro**. La réponse la plus courante est de répondre + ou - l'infini.
- **Inexact** : le résultat ne peut être représenté par un flottant et on doit l'arrondir.

Pour donner un exemple avec l'exception ***Inexact***, on va prendre le nombre 0.1. Ce nombre ne semble pourtant pas méchant, mais c'est parce qu'il est écrit en décimal. En binaire, ce nombre s'écrit comme ceci :
0 . 000 1100 1100 1100 1100 1100 1100... et ainsi de suite jusqu'à l'infini. Notre nombre utilise une infinité de décimales. Bien évidement, on ne peut pas utiliser une infinité de bits pour stocker notre nombre et on doit impérativement l'arrondir.

Comme vous le voyez avec la dernière exception, le codage des nombres flottants peut parfois poser problème : dans un ordinateur, il se peut qu'une opération sur deux nombres flottants donne un résultat qui ne peut être codé par un flottant. On est alors obligé d'arrondir ou de tronquer le résultat de façon à le faire rentrer dans un flottant. Pour éviter que des ordinateurs différents utilisent des méthodes d'arrondis différentes, on a décidé de normaliser les calculs sur les nombres flottants et les méthodes d'arrondis. Pour cela, la norme impose le support de quatre modes d'arrondis :

- **Arrondir vers + l'infini**,
- **vers - l'infini**,
- **vers zéro**
- **vers le nombre flottant le plus proche**.

Codage du texte

Nous savons donc comment faire pour représenter des nombres dans notre ordinateur, et c'est déjà un bon début. Mais votre ordinateur peut parfaitement manipuler autre chose que des nombres. Il peut aussi manipuler des images, du son, ou pleins d'autres choses encore. Eh bien sachez que tout cela est stocké dans votre ordinateur... sous la forme de nombres codés en binaire, avec uniquement des 0 et des 1.

Le **codage** définit la façon de représenter une information (du texte, de la vidéo, du son...) avec des nombres. Ce codage va attribuer à un nombre : une lettre, la couleur d'un pixel à l'écran... Ainsi, notre ordinateur sera non seulement capable de manipuler des nombres (et de faire des calculs avec), mais il sera aussi capable de manipuler une information ne représentant pas forcément un nombre pour l'utilisateur.

Bien évidement, l'ordinateur n'a aucun moyen de faire la différence entre un nombre qui code un pixel, un nombre qui code une lettre ou même un nombre. Pour lui, tout n'est que suites de zéro et de uns sans aucune signification : une donnée en binaire ne contient aucune information sur l'information qu'elle code (son "type"), et l'ordinateur n'a aucun moyen de le deviner.

Par exemple, si je vous donne la suite de bits suivante : 1100101 codé sur 7 bits ; vous n'avez aucun moyen de savoir s'il s'agit d'une lettre (la lettre e avec l'encodage ASCII), le nombre 101, ou l'entier -26 codé en complément à 1, ou encore l'entier -25 codé en complément à deux.

Ce qui va faire la différence entre les types c'est la façon dont sera interprétée la donnée : on n'effectuera pas les mêmes traitements sur une suite de bits selon ce qu'elle représente. Par exemple, si on veut afficher un 'e' à l'écran, les manipulations effectuées ne seront pas les mêmes que celles utilisée pour afficher le nombre 101, ou le nombre -25, etc.

Pour la suite, on va prendre l'exemple du texte.

Standard ASCII

Pour stocker un texte, rien de plus simple : il suffit de savoir comment stocker une lettre dans notre ordinateur et le reste coule de source. On va donc devoir coder chaque lettre et lui attribuer un nombre. Pour cela, il existe un standard, nommée la **table ASCII** qui va associer un nombre particulier à chaque lettre. L'ASCII est un standard qui permet à tous les ordinateurs de coder leurs caractères de la même façon. Ce standard ASCII utilise des nombres codés sur 7bits, et peut donc coder 128 symboles différents.

Notre table ASCII est donc une table de correspondance qui attribue un nombre à chaque symbole. La voici dans son intégralité, rien que pour vous.

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Si vous lisez en entier la table ASCII, vous remarquerez sûrement qu'il n'y a pas que des lettres codées par l'ASCII : il y tous les caractères d'un clavier qui sont inscrits dans cette table.

On peut faire quelques remarques sur cette table ASCII :

- Les lettres sont stockées dans l'ordre alphabétique, pour simplifier la vie des utilisateurs.
- Le passage des minuscules aux majuscules se fait en changeant le 6éme bit du caractère, ce qui est très simple.
- Les symboles 0 à 31 , ainsi que le 127éme sont un peu bizarres...

Ces symboles présents dans ce standard ASCII ne peuvent même pas être tapés au clavier et ils ne sont pas affichables !

Mais à quoi peuvent-ils bien servir ?

Il faut savoir que ce standard est assez ancien. A l'époque de la création de ce standard, il existait de nombreuses imprimantes et autres systèmes qui l'utilisaient Et pour faciliter la conception de ces machines, on a placé dans cette table ASCII des symboles qui n'étaient pas destinés à être affichés, mais dont le but était de donner un ordre à l'imprimante/machine à écrire... On trouve ainsi des symboles de retour à la ligne, par exemple.

Unicode

Le problème avec la table ASCII, c'est qu'on se retrouve assez rapidement limité avec nos 128 symboles. On n'arrive pas à caser les accents ou certains symboles particuliers à certaines langues dedans. Impossible de coder un texte en grec ou en japonais : les idéogrammes et les lettres grecques ne sont pas dans la table ASCII. Pour combler ce genre de manque, de nombreuses autres méthodes sont apparues qui peuvent coder bien plus de symboles que la table ASCII. Elles utilisent donc plus de 7 bits pour coder leurs symboles : on peut notamment citer l'unicode. Pour plus de simplicité, l'unicode est parfaitement compatible avec la table ASCII : les 128 premiers symboles de l'unicode sont ceux de la table ASCII, et sont rangés dans le même ordre.

Si vous voulez en savoir plus sur ces encodages, sachez qu'il existe un tutoriel sur le sujet, sur le siteduzéro. Le voici : Comprendre les encodages.

Nos bits prennent la porte !

Grâce au chapitre précédent, on sait enfin comment sont représentées nos données les plus simples avec des bits. On n'est pas encore allés bien loin : on ne sait pas comment représenter des bits dans notre ordinateur ou les modifier, les manipuler, ni faire quoi que ce soit avec. On sait juste transformer nos données en paquets de bits (et encore, on ne sait vraiment le faire que pour des nombres entiers, des nombres à virgule et du texte...). C'est pas mal, mais il reste du chemin à parcourir ! Rassurez-vous, ce chapitre est là pour corriger ce petit défaut. On va vous expliquer comment représenter des bits dans un ordinateur et quels traitements élémentaires notre ordinateur va effectuer sur nos bits. Et on va voir que tout cela se fait avec de l'électricité !

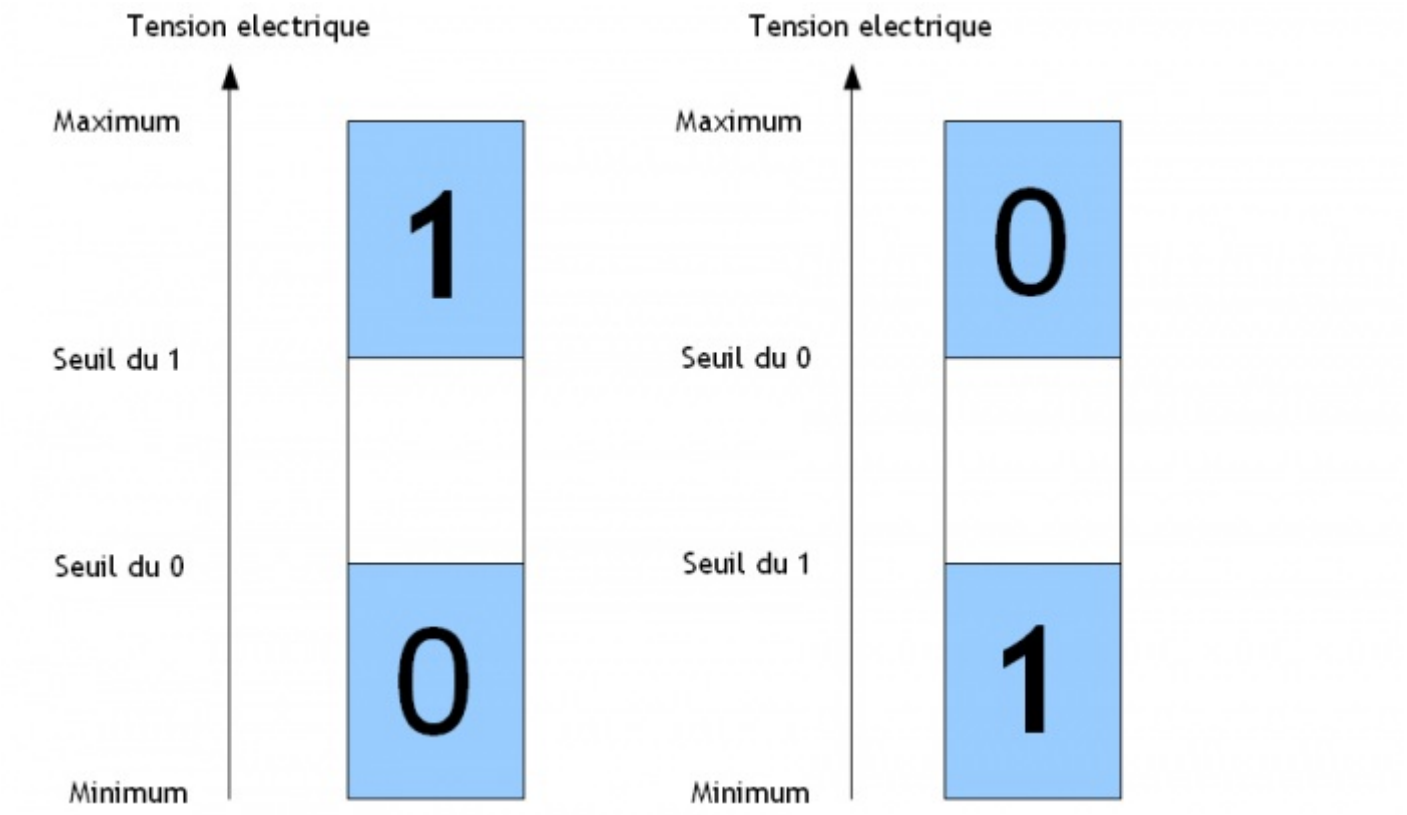
Codage NRZ

Pour compter en binaire , il faut travailler avec des bits qui peuvent prendre deux valeurs notées 0 et 1. Le tout est de savoir comment représenter ces bits dans l'ordinateur. Pour cela, on utilise une grandeur physique nommée la tension. Pas besoin de savoir ce que c'est, sachez juste que ça se mesure en volts et que ça n'est pas synonyme de courant électrique. Rien à voir avec un quelconque déplacement d'électrons, comme certains le pensent.

Avec cette tension, il y a diverses méthodes pour coder un bit : codage Manchester, NRZ, etc. Ces diverses méthodes ont chacune leurs avantages et leurs défauts. Autant trancher dans le vif tout de suite : la quasi-intégralité des circuits de notre ordinateur se basent sur le **codage NRZ**.

Codage NRZ

Pour coder un 0 ou 1 en NRZ, si suffit de dire que si la tension est en-dessous d'un seuil donné, C'est un 0. Et il existe un autre seuil au-dessus duquel la tension représente un 1. Du moins, c'est ainsi dans la majorité des cas : il arrive que ce soit l'inverse sur certains circuits élèctroniques : en-dessous d'un certain seuil, c'est un 1 et si c'est au-dessus d'un autre seuil c'est 0. Tout ce qu'il faut retenir, c'est qu'il y a un intervalle pour le 0 et un autre pour le 1. En dehors de ces intervalles, on considère que le circuit est trop imprécis pour pouvoir conclure sur la valeur de la tension : on ne sait pas trop si c'est un 1 ou un 0.



Il y a deux seuils, car les circuits qui manipulent des tensions n'ont pas une précision parfaite, et qu'une petite perturbation électrique pourrait alors transformer un 0 en 1. Pour limiter la casse, on préfère séparer ces deux seuils par une sorte de marge de sécurité.

Tensions de référence

Ces tensions vont être manipulées par différents circuits électroniques plus ou moins sophistiqués. Pour pouvoir travailler avec des tensions, nos circuits ont besoin d'être alimentés en énergie. Pour cela, notre circuit possédera une tension qui alimentera le

circuit en énergie, qui s'appelle la **tension d'alimentation**. Après tout, si un circuit doit coder des bits valant 1, il faudra bien qu'il trouve de quoi fournir une tension de 2, 3, 5 volts : la tension codant notre 1 ne sort pas de nulle part ! De même, on a besoin d'une tension de référence valant zéro volt, qu'on appelle la **masse**, qui sert pour le zéro.

Dans tous les circuits électroniques (et pas seulement les ordinateurs), cette tension d'alimentation varie généralement entre 0 et 5 volts. Mais de plus en plus, on tend à utiliser des valeurs de plus en plus basses, histoire d'économiser un peu d'énergie. Et oui, car plus un circuit utilise une tension élevée, plus il consomme d'énergie et plus il chauffe.

Pour un processeur, il est rare que les modèles récents utilisent une tension supérieure à 2 volts : la moyenne tournant autour de 1-1.5 volts. Même chose pour les mémoires : la tension d'alimentation de celle-ci diminue au court du temps. Pour donner des exemples, une mémoire DDR a une tension d'alimentation qui tourne autour de 2,5 volts, les mémoires DDR2 ont une tension d'alimentation qui tombe à 1,8 volts, et les mémoires DDR3 ont une tension d'alimentation qui tombe à 1,5 volts. C'est très peu : les composants qui manipulent ces tensions doivent être très précis.

Transistors

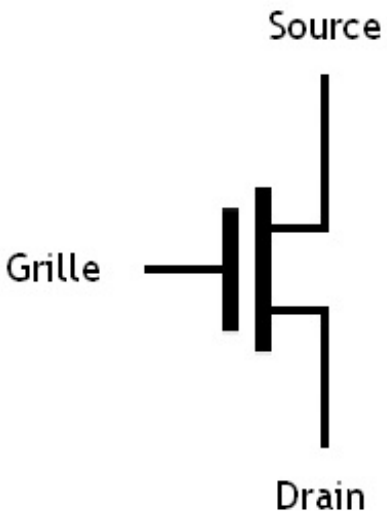
Pour commencer, nous allons devoir faire une petite digression et parler un peu d'électronique : sans cela, impossible de vous expliquer en quoi est fait un ordinateur ! Sachez tout d'abord que nos ordinateurs sont fabriqués avec des composants électroniques que l'on appelle des **transistors**, reliés pour former des circuits plus ou moins compliqués. Presque tous les composants d'un ordinateur sont fabriqués avec un grand nombre de transistors, qui peut monter à quelques milliards sur des composants sophistiqués. Pour donner un exemple, sachez que les derniers modèles de processeurs peuvent utiliser près d'un milliard de transistors. Et le tout doit tenir sur quelques centimètres carrés : autant vous dire que la miniaturisation a fait d'énormes progrès !

Transistors CMOS

Il existe différents types de transistors, chacun avec ses particularités, ses avantages et ses inconvénients. On ne va pas en parler plus que ça, mais il faut préciser que les transistors utilisés dans nos ordinateurs sont des transistors à effet de champ à technologie **CMOS**. Si vous ne comprenez pas ce que ça signifie, ce n'est pas grave, c'est un simple détail sans grande importance.

Mais qu'est-ce qu'un transistor **CMOS** ?

Il s'agit simplement d'un composant relié à un circuit électronique par trois morceaux de "fil" conducteur que l'on appelle **broches**. On peut appliquer de force une tension électrique sur ces broches (attention à ne pas la confondre avec le courant électrique), qui peut représenter soit **0** soit **1** en fonction du transistor utilisé.



Ces trois broches ont des utilités différentes et on leur a donné un nom pour mieux les repérer :

- la grille ;
- le drain ;
- la source.

Dans les processeurs, on utilise notre transistor comme un interrupteur qui réagit en fonction de sa grille : suivant la valeur de la tension qui est appliquée sur la grille, le transistor conduira ou ne conduira pas le courant entre la source et le drain. En clair, appliquez la tension adéquate et la liaison entre la source et le drain se comportera comme un interrupteur fermé et conduira le courant : le transistor sera alors dit dans **l'état passant**. Par contre, si vous appliquez une tension à la bonne valeur sur la grille, cette liaison se comportera comme un interrupteur ouvert et le courant ne passera pas : le transistor sera dit dans **l'état bloqué**.

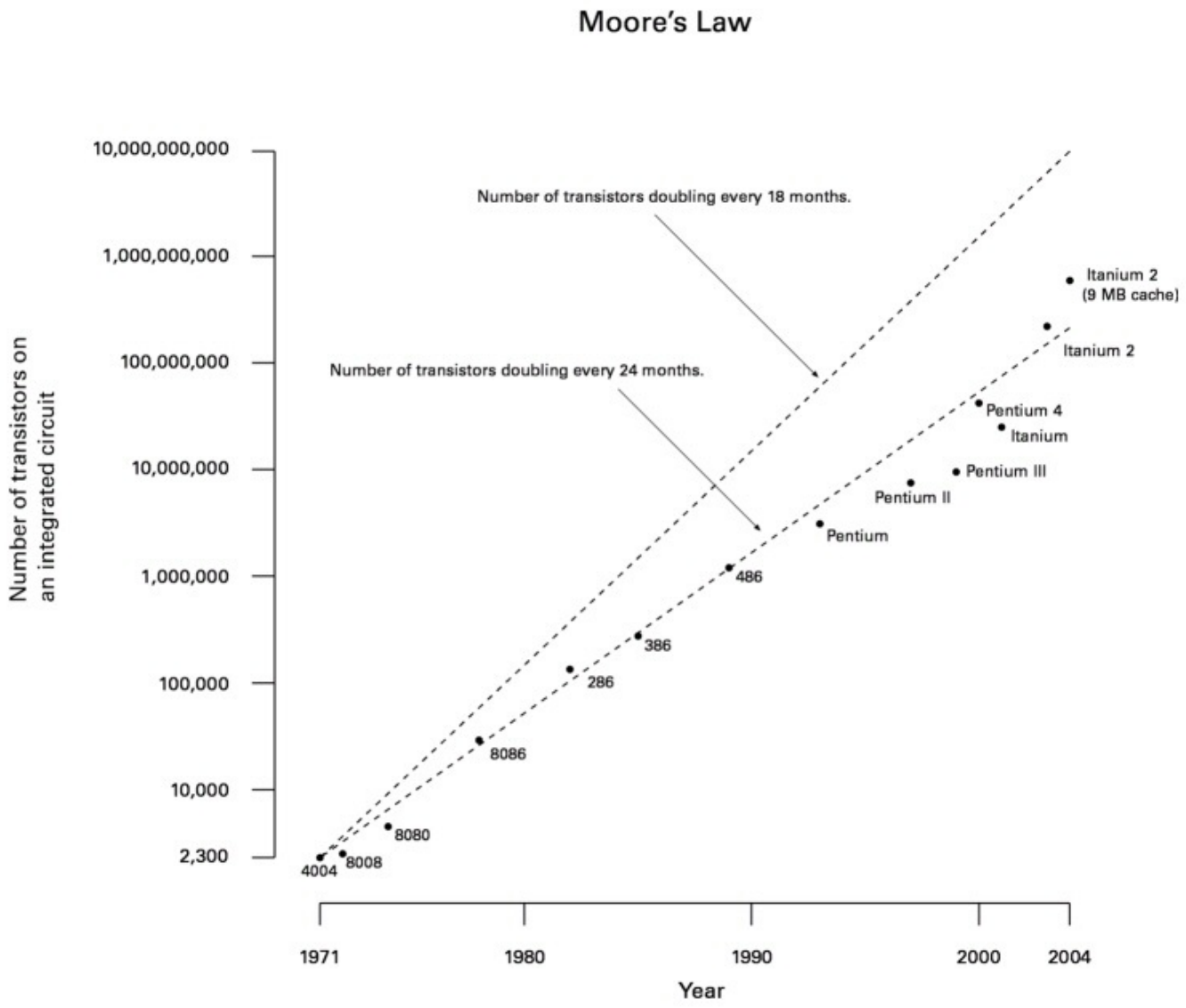
Il existe deux types de transistors CMOS, qui diffèrent entre autres par la tension qu'il faut mettre sur la grille pour les ouvrir/fermer :

- les transistors **NMOS** qui s'ouvrent lorsqu'on place une tension égale à zéro sur la grille et se ferment si la tension placée sur cette même grille représente un 1 ;
- et les **PMOS** pour qui s'est l'inverse : ils se ferment lorsque la tension sur la grille est nulle, et s'ouvrent si celle-ci représente un 1.

Loi de Moore

De nos jours, le nombre de transistors des composants électroniques actuels augmente de plus en plus, et les concepteurs de circuits rivalisent d'ingéniosité pour miniaturiser le tout.

En 1965, le cofondateur de la société Intel, spécialisée dans la conception des mémoires et de processeurs, a affirmé que la quantité de transistors présents dans un processeur doublait tous les 18 mois. Cette affirmation porte aujourd'hui le nom de **première loi de Moore**. En 1975, le cofondateur d'Intel réévalua cette affirmation : ce n'est pas tous les 18 mois que le nombre de transistors d'un processeur double, mais tous les 24 mois. Cette nouvelle version, appelée la seconde loi de Moore, a redoutablement bien survécue : elle est toujours valable de nos jours.



Ce faisant, la complexité des processeurs augmente de façon exponentielle dans le temps et sont censés devenir de plus en plus gourmands en transistors au fil du temps.

De plus, miniaturiser les transistors permet parfois de les rendre plus rapides : c'est un scientifique du nom de Robert Dennard qui a découvert un moyen de rendre un transistor plus rapide en diminuant certains paramètres physiques d'un transistor. Sans cette miniaturisation, vous pouvez être certains que nos processeurs en seraient pas aussi complexes qu'aujourd'hui. Mais attention, cela ne signifie pas pour autant que le nombre de transistors soit un indicateur efficace de performances : avoir beaucoup de transistors ne sert à rien si on le les utilise pas correctement.

Mais cette miniaturisation a ses limites et elle pose de nombreux problèmes dont on ne parlera pas ici. Sachez seulement que cette loi de Moore restera valable encore quelques dizaines d'années, et qu'au delà, on ne pourra plus rajouter de transistors dans nos processeurs aussi facilement que de nos jours.

Portes logiques

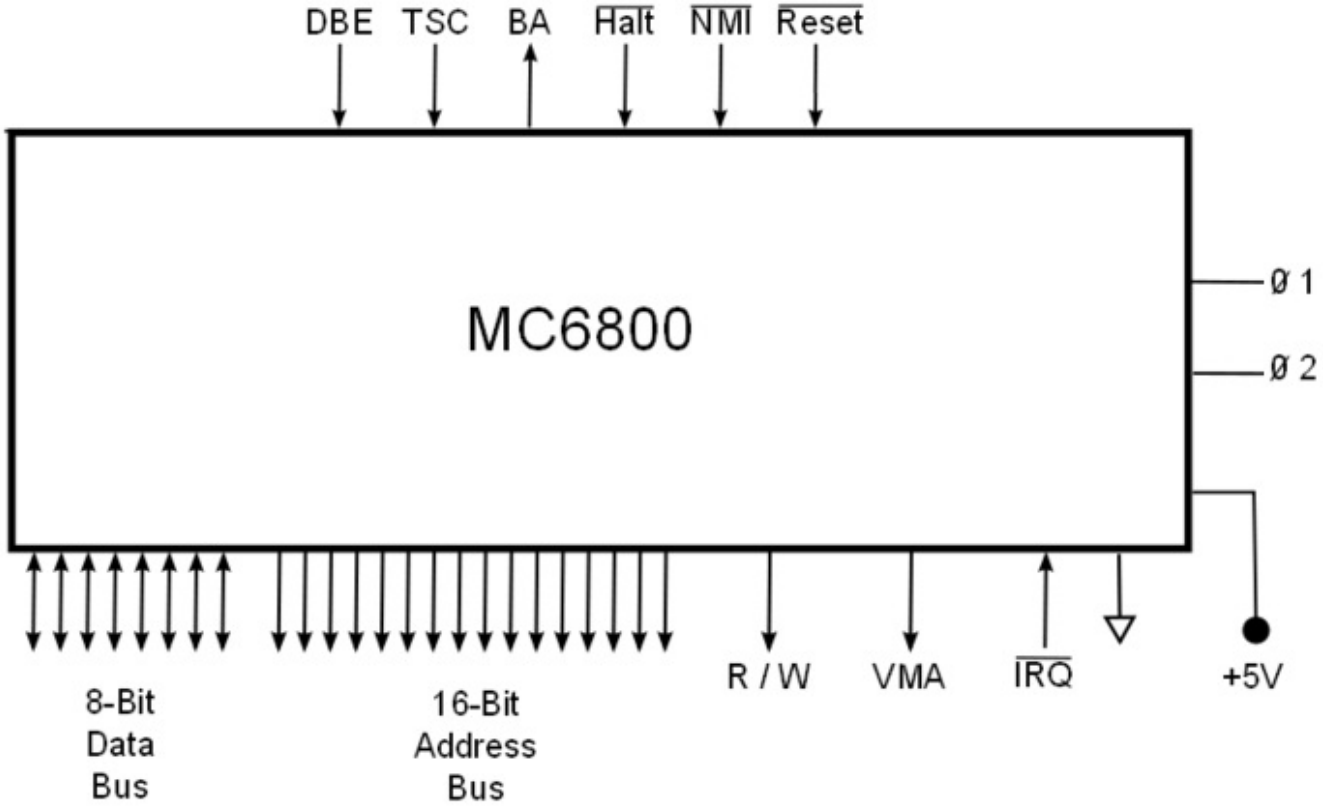
C'est bien beau de savoir coder des bits et d'avoir des transistors pour les manipuler, mais j'aimerais savoir comment on fait pour triturer des bits avec des transistors ?

Et bien que vos vœux soient exhaussés ! La solution consiste à rassembler ces transistors dans ce qu'on appelle des **circuits logiques**.

Ce sont simplement des petits circuits, fabriqués avec des transistors, qui possèdent des sorties et des entrées, sur lesquelles on va placer des bits pour les manipuler. Ces entrées et ces sorties ne sont rien d'autre que des morceaux de "fil" conducteur sur lesquelles on peut mesurer une tension qui représente un zéro ou un 1. Sur chaque entrée du composant, on peut forcer la valeur

de la tension, histoire de mettre l'entrée à 0 ou à 1. A partir de là, le circuit électronique va réagir et déduire la tension à placer sur chacune de ses sorties en fonction de ses entrées.

Autant vous le dire tout de suite, votre ordinateur est remplit de ce genre de choses. Quasiment tous les composants de notre ordinateur sont fabriqués avec ce genre de circuits. Par exemple, notre processeur est un composant électronique comme un autre, avec ses entrées et ses sorties.



Brochage d'un processeur MC68000.

L'exemple montré au dessus est un processeur MC68000, un vieux processeur, présent dans les calculatrices TI-89 et TI-92, qui contient 68000 transistors (d'où son nom : MC68000) et inventé en 1979. Il s'agit d'un vieux processeur complètement obsolète et particulièrement simple. Et pourtant, il y en a des entrées et des sorties : 37 au total ! Pour comparer, sachez que les processeurs actuels utilisent entre 700 et 1300 broches d'entrée et de sortie. A ce jeu là, notre pauvre petit MC68000 passe pour un gringalet !

Le nombre de broches (entrées et sorties) d'un processeur dépend du socket de la carte mère. Par exemple, un socket LGA775 est conçu pour les processeurs comportant 775 broches d'entrée et de sortie, tandis qu'un socket AM2 est conçu pour des processeurs de 640 broches. Certains sockets peuvent carrément utiliser 2000 broches (c'est le cas du socket G34 utilisé pour certains processeurs AMD Opteron).

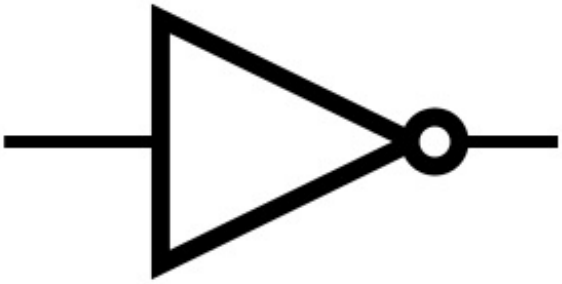
Pour la mémoire, le nombre de broches dépend du format utilisé pour la barrette de mémoire (il existe trois formats différents), ainsi que du type de mémoire. Certaines mémoires obsolètes (les mémoires FPM-RAM et EDO-RAM) se contentaient de 30 broches, tandis que la mémoire DDR2 utilise entre 204 et 244 broches.

Néanmoins, quelque soit la complexité du circuit à créer, celui-ci peut être construit en reliant quelques petits circuits de base entre eux. Ces circuits de base sont nommés des **portes logiques**. Il existe trois portes logiques qui sont très importantes et que vous devez connaitre : les portes ***ET***, ***OU*** **et** ***NON***. Mais pour se faciliter la vie, on peut utiliser d'autres portes, plus ou moins différentes. Voyons un peu quelles sont ces portes, et ce qu'elles font.

La porte NON

Le premier opérateur fondamental est la porte ***NON*** aussi appelée porte inverseuse. Cette porte agit sur un seul bit.

Elle est symbolisée par le schéma suivant :



Sur les schémas qui vont suivre, les entrées des portes logiques seront à gauche et les sorties à droite !

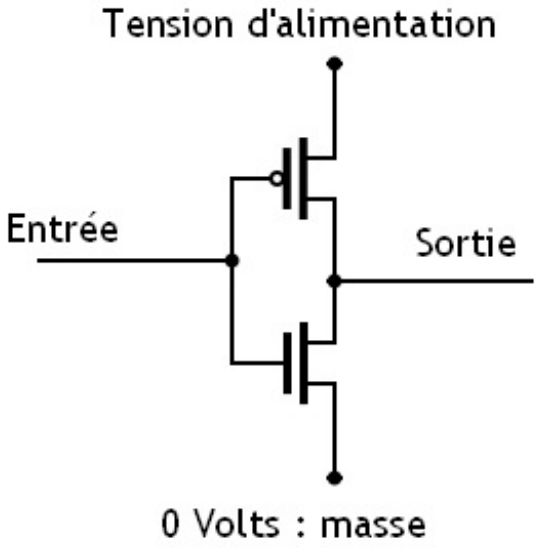
Pour simplifier la compréhension, je vais rassembler les états de sortie en fonction des entrées pour chaque porte logique dans un tableau qu'on appelle table de vérité. Voici celui de la porte ***NON*** :

Entrée	Sortie
0	1
1	0

Le résultat est très simple, la sortie d'une porte ***NON*** est exactement le contraire de l'entrée.

Câblage

Cette porte est fabriquée avec seulement deux transistors et son schéma est diablement simple. Voici le montage en question.



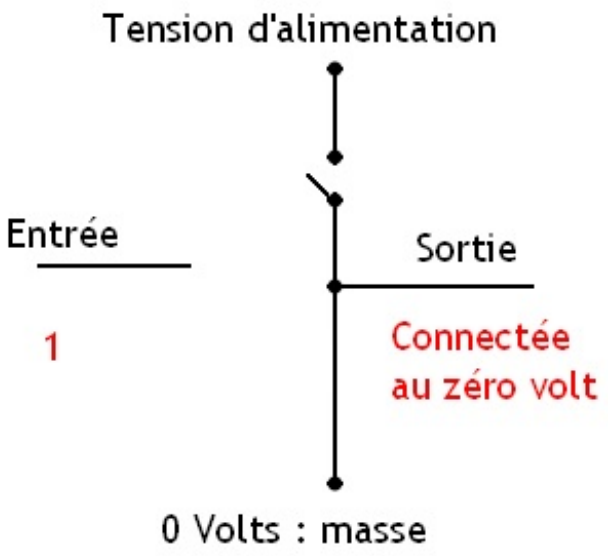
Je crois que çà mérite une petite explication, non ?

Rappelez-vous qu'un transistor CMOS n'est rien d'autre qu'un interrupteur, qu'on peut fermer suivant ce qu'on met sur sa grille. Certains transistors se ferment quand on place un 1 sur la grille, et d'autres quand on place un zéro.

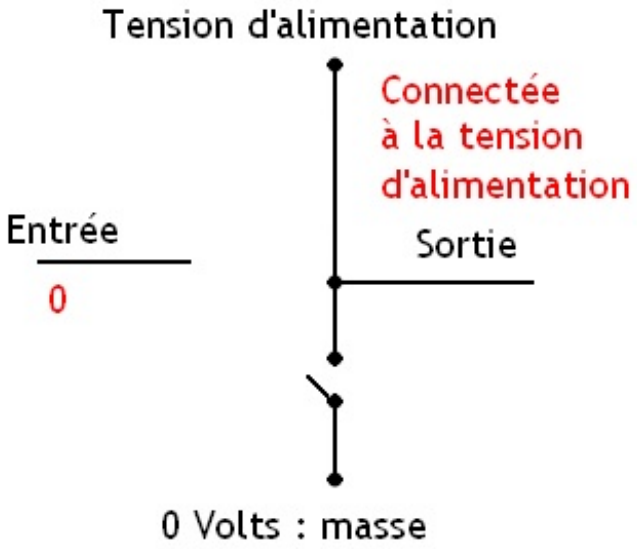
L'astuce du montage vu plus haut consiste à utiliser deux transistors différents :

- celui du haut conduit le courant quand on place un 0 sur sa grille, et ne conduit pas le courant sinon ;
- et celui du bas fait exactement l'inverse.

Si on met un 1 en entrée de ce petit montage électronique, le transistor du haut va fonctionner comme un interrupteur ouvert, et celui du bas comme un interrupteur fermé. On se retrouvera donc avec notre sortie reliée au zéro volt, et donc qui vaut zéro.



Inversement, si on met un 0 en entrée de ce petit montage électronique, le transistor du bas va fonctionner comme un interrupteur ouvert, et celui du haut comme un interrupteur fermé. On se retrouvera donc avec notre sortie reliée à la tension d'alimentation, qui vaudra donc 1.



Comme vous le voyez, avec un petit nombre de transistors, on peur réussir à créer de quoi inverser un bit. Et on peut faire pareil avec toutes les autres portes élémentaires : on prend quelques transistors, on câble cela comme il faut, et voilà une porte logique toute neuve !

La porte ET

Maintenant une autre porte fondamentale : la porte ***ET***.

Cette fois, différence avec la porte *NON*, la porte ***ET*** a 2 entrées, mais une seule sortie.

Voici comment on la symbolise :

Cette porte a comme table de vérité :

Entrée 1	Entrée 2	Sortie
0	0	0
0	1	0
1	0	0
1	1	1

Cette porte logique met sa sortie à 1 quand toutes ses entrées valent 1.

Porte NAND

La porte ***NAND*** est l'exact inverse de la sortie d'une porte ***ET***. Elle fait la même chose qu'une porte ***ET*** suivie d'une porte ***NON***.

Sa table de vérité est :

Entrée 1	Entrée 2	Sortie
0	0	1
0	1	1
1	0	1
1	1	0

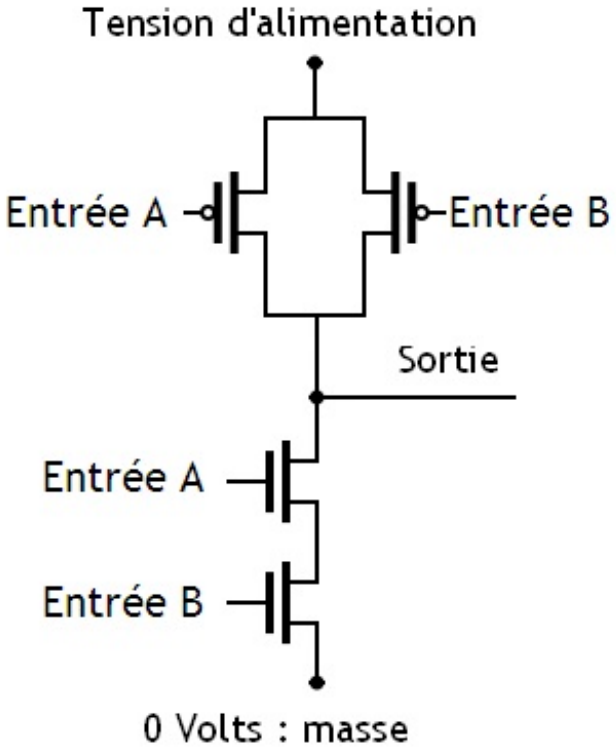
Cette porte a une particularité : on peut recréer les portes ***ET***, ***OU*** et ***NON***, et donc n'importe quel circuit électronique, en utilisant des montages composés uniquement de portes ***NAND***. A titre d'exercice, vous pouvez essayez de recréer les portes ***ET***, ***OU*** et ***NON*** à partir de portes ***NAND***. Ce serait un petit entrainement assez sympathique. Après tout, si ça peut vous occuper lors d'un dimanche pluvieux.

On la symbolise par le schéma qui suit.

Au fait, si vous regardez le schéma de la porte ***NAND***, vous verrez que son symbole est presque identique à celui d'une porte ***ET*** : seul le petit rond ajouté sur la sortie de la porte a été rajouté. Et bien sachez que ce petit rond est une sorte de raccourci pour schématiser une porte ***NON***. Ainsi, si vous voyez un petit rond quelque part sur un schéma (sur une entrée ou une sortie, peut importe), sachez que ce petit rond symbolise une porte ***NON***.

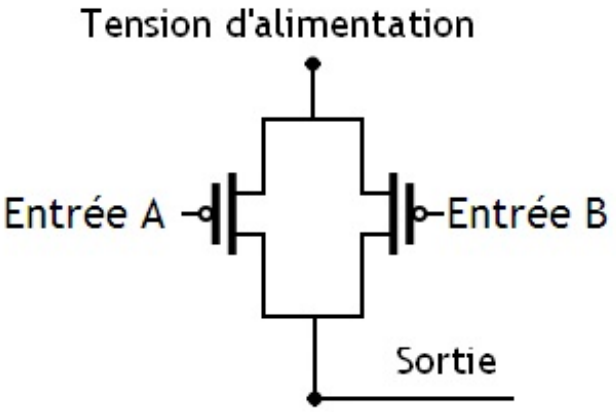
Câblage

Implémenter une porte ***NAND*** avec des transistors CMOS est un peu plus complexe qu'implémenter une porte ***NON***. Mais qu'à cela ne tienne, voici en exclusivité : comment créer une porte ***NAND*** avec des transistors CMOS !



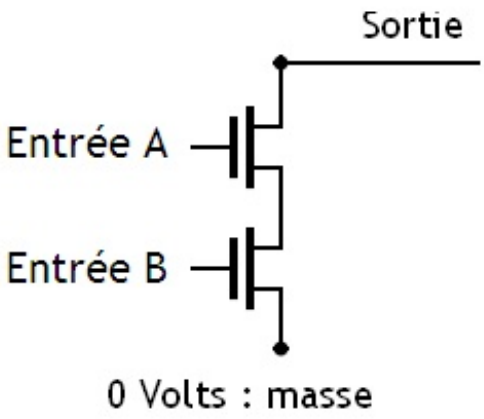
Ce schéma peut s'expliquer très simplement. Tout d'abord, vous verrez qu'il y a deux grands blocs de transistors dans ce circuit : un entre la sortie et la tension d'alimentation, et un autre entre la sortie et la masse. Tous les circuits CMOS suivent ce principe, sans exception. Ensuite, on peut remarquer que tous les transistors placés entre la tension d'alimentation et la sortie sont des transistors PMOS. De même, tous les transistors placés entre la masse et la sortie sont des transistors NMOS. Ceci est encore une fois vrai pour tous les circuits CMOS.

Regardons ces deux parties l'une après l'autre, en commençant par celle du haut.



Celle-ci sert à connecter la sortie sur la tension d'alimentation du circuit. Nos deux transistors sur de type PMOS : ils se ferment quand on leur met un 0 sur la grille. Or, les transistors sont mis en parallèle : si un seul de ces deux transistors est fermé, la tension d'alimentation sera reliée à la sortie et elle passera à 1. Donc, si une seule des deux entrées est à 0, on se retrouve avec un 1 en sortie.

Passons maintenant à l'autre bloc de transistors.



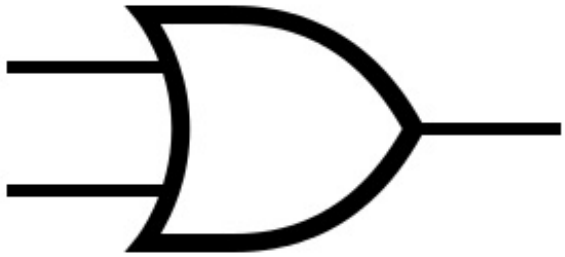
Cette fois-ci, c'est l'inverse : nos transistors sont reliés les uns à la suite des autres : il faut que les deux soient fermés pour que la masse soit connectée à la sortie. Et les transistors sont cette fois des transistors NMOS : ils se ferment quand on leur met un 1 sur leur grille. Donc, pour avoir un zéro en sortie, il faut que les deux entrées soient à 1. Au final on obtient bien une porte NAND.

La porte OU

Maintenant une autre porte fondamentale : la porte ***OU***.

Cette fois, comme la porte ***ET***, elle possède 2 entrées, mais une seule sortie.

On symbolise cette porte comme ceci :



Cette porte est définie par la table de vérité suivante :

Entrée 1	Entrée 2	Sortie
0	0	0
0	1	1
1	0	1
1	1	1

Cette porte logique met sa sortie à 1 quand au moins une de ses entrées vaut 1.

Porte NOR

La porte ***NOR*** est l'exact inverse de la sortie d'une porte ***OU***. Elle est équivalente à une porte ***OU*** suivie d'une porte ***NON***.

Sa table de vérité est :

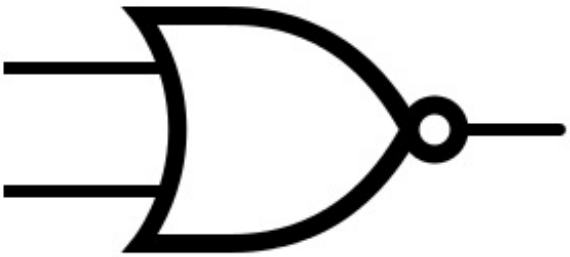
Entrée 1	Entrée 2	Sortie

0	0	1
0	1	0
1	0	0
1	1	0

On peut recréer les portes ***ET***, ***OU*** et ***NON***, et donc n'importe quel circuits électronique, en utilisant des montages composés uniquement de portes ***NOR***. Comme quoi, la porte ***NAND*** n'est pas la seule à avoir ce privilège. Cela a une conséquence : on peut concevoir un circuits en n'utilisant que des portes ***NOR***. Pour donner un exemple, sachez que les ordinateurs chargés du pilotage et de la navigation des missions Appollo étaient intégralement conçus uniquement avec des portes ***NOR***.

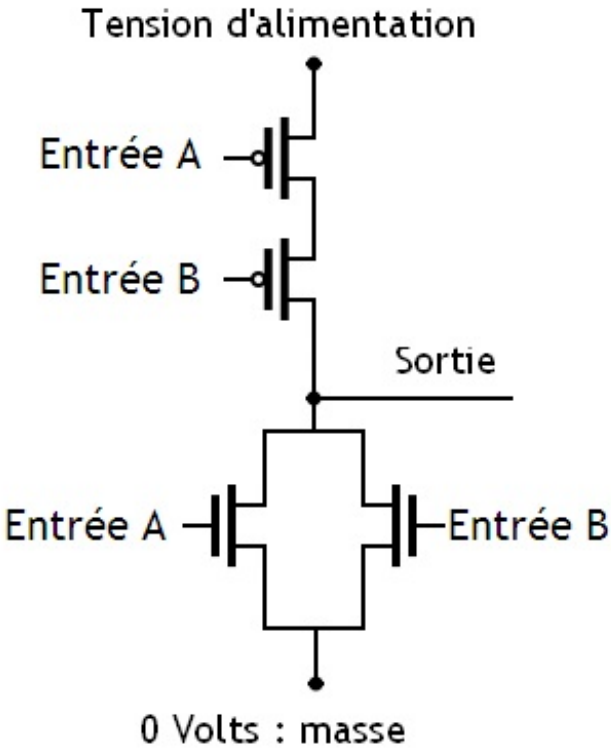
A titre d'exercice, vous pouvez essayez de recréer les portes ***ET***, ***OU*** et ***NON*** à partir de portes ***NOR***. Si vous en avez envie, hein !

On la symbolise avec le schéma qui suit.

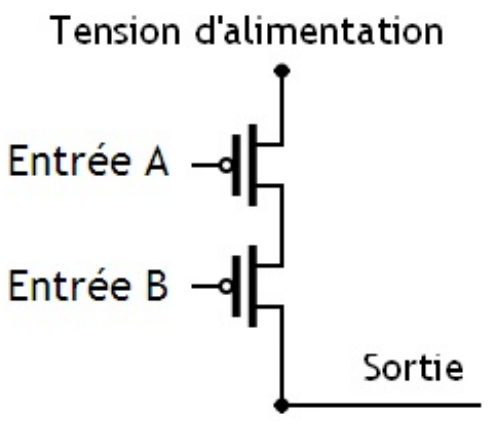


Câblage

Implémenter une porte ***NOR*** avec des transistors CMOS ressemble à ce qu'on a fait pour la prote ***NAND***.

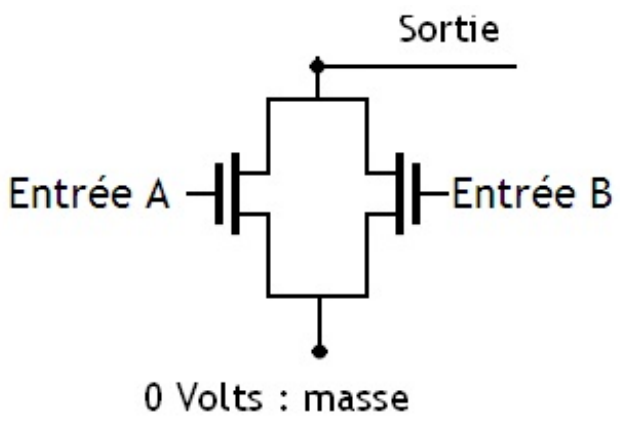


Ce schéma peut s'expliquer très simplement. Encore une fois, on va voir chacune des deux parties (celle du haut et celle du bas) l'une après l'autre, en commençant par celle du haut.



Celle-ci sert à connecter la sortie sur la tension d'alimentation du circuit. Nos deux transistors sur de type PMOS : ils se ferment quand on leur met un 0 sur la grille. Nos transistors sont reliés les uns à la suite des autres : il faut que les deux soient fermés pour que la masse soit connectée à la sortie. les deux entrées doivent être à zéro pour que l'on ait un 1 en sortie.

Passons maintenant à l'autre bloc de transistors.



Les transistors sont des transistors NMOS : ils se ferment quand on leur met un 1 sur leur grille. Cette fois, les transistors sont mis en parallèle : si un seul de ces deux transistors est fermé, la tension d'alimentation sera reliée à la sortie et elle passera à 0. Donc, si une seule des deux entrées est à 1, on se retrouve avec un 1 en sortie. Au final on obtient bien une porte NOR.

Porte XOR

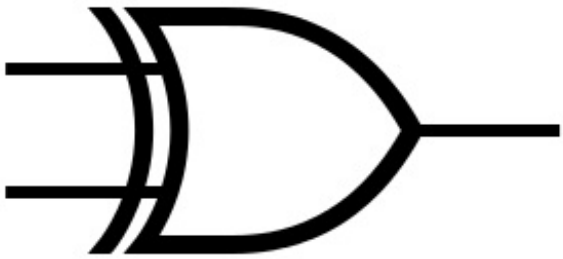
Avec une porte ***OU*** , deux ***ET*** et deux portes ***NON***, on peut créer une porte nommée **XOR**. Cette porte est souvent appelée porte ***OU Exclusif*** .

Sa table de vérité est :

Entrée 1	Entrée 2	Sortie
0	0	0
0	1	1
1	0	1
1	1	0

On remarque que sa sortie est à 1 quand les deux bits placés sur ses entrées sont différents, et valent 0 sinon.

On la symbolise comme ceci :



Porte NXOR

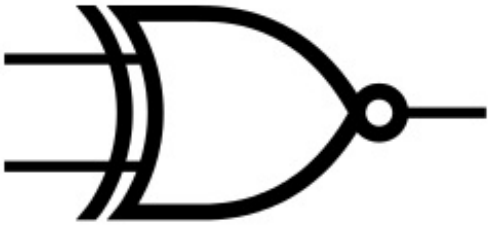
La porte **XOR** posséde une petite soueur : la ***NXOR***.

Sa table de vérité est :

Entrée 1	Entrée 2	Sortie
0	0	1
0	1	0
1	0	0
1	1	1

On remarque que sa sortie est à 1 quand les deux bits placés sur ses entrées sont différents, et valent 0 sinon. Cette porte est équivalente à une porte ***XOR*** suivie d'une porte ***NON***.

On la symbolise comme ceci :



Créons nos circuits !

Bon, c'est bien beau d'avoir quelques portes logiques, mais si je veux créer un circuit, je fais comment ?

Il faut avouer qu'on irait pas loin en sachant uniquement ce que sont les ***ET***, ***NAND***, et autres. Ce qu'il faudrait, c'est pouvoir créer de vrais circuits. Et bien que vos vœux soient exaucés (enfin presque) : nous allons enfin voir comment sont réalisés les circuits de nos ordinateurs. Du moins, nous allons voir comment créer des circuits simples, mais qui sont à la base des circuits de notre ordinateur.

Circuits combinatoires

Pour commencer, nous allons parler d'une classe de circuits assez simples : les **circuits combinatoires**. Ces circuits font comme tous les autres circuits : ils prennent des données sur leurs entrées, et fournissent un résultat en sortie. Le truc, c'est que ce qui est fourni en sortie ne dépend que du résultat sur les entrées, et de rien d'autre ! Cela peut sembler être évident, mais on verra que ce n'est pas le cas pour tous les circuits.

Pour donner quelques exemples de circuits combinatoires, on peut citer les circuits qui effectuent des additions, des multiplications, ou d'autres opérations arithmétiques du genre. Par exemple, le résultat d'une addition ne dépend que des nombres à additionner et rien d'autre. Pareil pour la division, la soustraction, la multiplication, etc. Notre ordinateur contient de nombreux circuits de ce genre. Toutefois, nous ne verrons pas tout de suite les circuits capables d'effectuer ces calculs : ceux-ci sont un peu plus compliqués que ce qu'on va voir ici et on va donc les laisser pour plus tard, dans la partie sur le processeur.

Tables de vérité

Bref, poursuivons. J'ai promis de vous apprendre à concevoir des circuits, de façon "simple". Pour commencer, il va falloir décrire ce que notre circuit fait. Pour un circuit combinatoire, la tâche est très simple, vu que ce qu'on trouve sur ses sorties ne dépend que de ce qu'on a sur les entrées. Pour décrire intégralement le comportement de notre circuit, il suffit donc de lister la valeur de chaque sortie pour toute valeur possible en entrée. Cela peut se faire simplement en écrivant ce qu'on appelle la **table de vérité** du circuit. Pour créer cette table de vérité, il faut commencer par lister toutes les valeurs possibles des entrées dans un tableau, et écrire à coté les valeurs des sorties qui correspondent à ces entrées. Cela peut être assez long : pour un circuit ayant n entrées, ce tableau aura 2^n lignes.

Bit de parité

Pour donner un exemple, on va prendre l'exemple d'un circuit calculant la **le bit de parité** d'un nombre.

Le quoi ?

Ah oui, pardon !

Ce bit de parité est une technique qui permet de détecter des erreurs de transmission ou d'éventuelles corruptions de données qui modifient un nombre impair de bits. Si un, trois, cinq, ou un nombre impair de bits voient leur valeur s'inverser (un 1 devient un 0, ou inversement), l'utilisation d'un bit de parité permettra de détecter cette erreur. Par contre, il sera impossible de la corriger.

Le principe caché derrière un bit de parité est simple : il suffit d'ajouter un bit supplémentaire aux bits à stocker. Le but d'un bit de parité est de faire en sorte que le nombre de bits à 1 dans le nombre à stocker, bit de parité inclut, soit toujours un nombre pair. Ce bit, le bit de parité vaudra :

- zéro si le nombre de bits à 1 dans le nombre à stocker (bit de parité exclu) est pair ;
- 1 si ce nombre est impair.

Détecter une erreur est simple : on compte le nombre de bits à 1 dans le nombre à stocker, bit de parité inclut, et on regarde s'il est pair. S'il est impair, on sait qu'au moins un bit à été modifié.

Table de vérité du circuit

Dans notre cas, on va créer un circuit qui calcule le bit de parité d'un nombre de 3 bits. Celui-ci dispose donc de 3 entrées, et d'une sortie sur laquelle on retrouvera notre bit de parité. Notre tableau possédera donc 2^3 lignes : cela fait 8 lignes. Voici donc le tableau de ce circuit, réalisé ci-dessous.

Entrée e2	Entrée e1	Entrée e0	Sortie s0
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

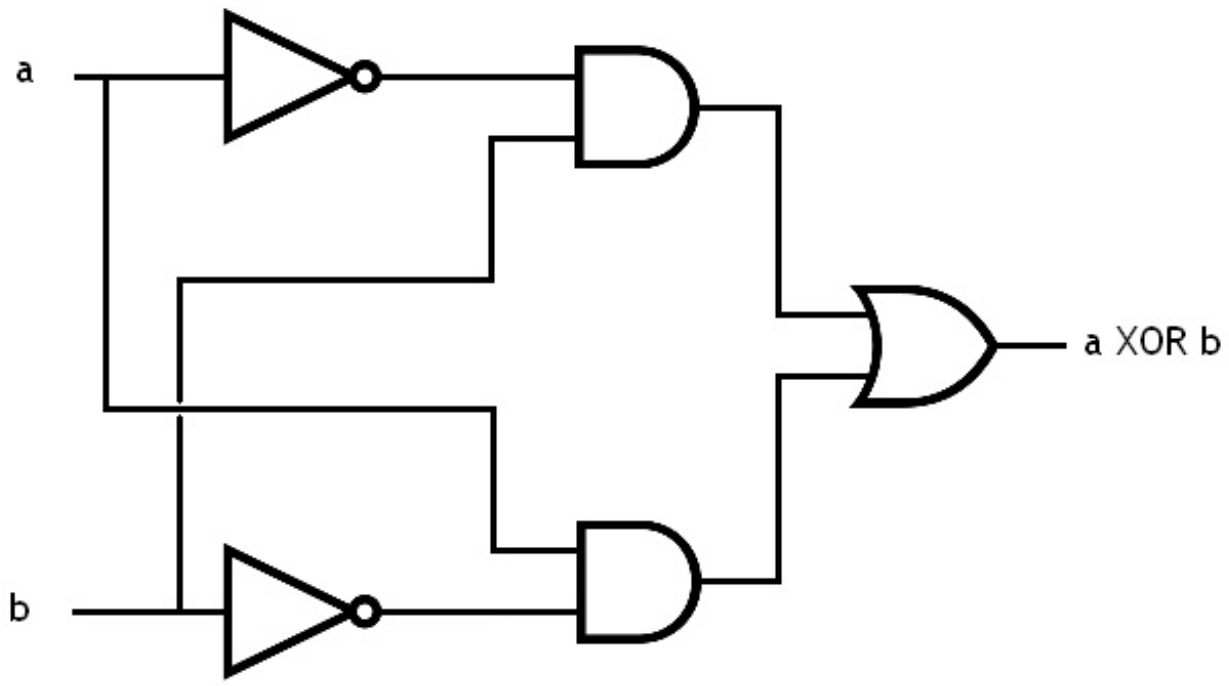
Équations logiques

Une fois qu'on a la table de vérité, une bonne partie du travail à déjà été fait. Il ne nous reste plus qu'à transformer notre table en ce qu'on appelle des **équations logiques**.

Mais que viennent faire les équations ici ?

Attention : il ne s'agit pas des équations auxquelles vous êtes habitués. Ces équations logiques ne font que travailler avec des 1 et des 0, et n'effectuent pas d'opérations arithmétiques mais seulement des ***ET***, des ***OU***, et des ***NON***. Ces équations vont ainsi avoir des bits pour inconnues.

Chacune de ces équations logiques correspondra à un circuit, et vice-versa : à un circuit sera associé une équation qui permettra de décrire le circuit. Par exemple, prenons le circuit vu dans le QCM de la question précédente.



Ce circuit a pour équation logique $(\overline{a}.b) + (a.\overline{b})$

Syntaxe

Pour pouvoir commencer à écrire ces équations, il va falloir faire un petit point de syntaxe. Voici résumé dans ce tableau les différentes opérations, ainsi que leur notation. Dans ce tableau, a et b sont des bits.

Opération logique	Symbole
NON a	$\overline{a}$
a ***ET*** b	$a.b$
a ***OU*** b	$a+b$
a ***XOR*** b	$a \oplus b$

Voilà, avec ce petit tableau, vous savez comment écrire une équation logique...enfin presque, il ne faut pas oublier le plus important : les parenthèses ! Et oui, il faudra bien éviter quelques ambiguïtés dans nos équations. C'est un peu comme avec des équations normales : $(a \times b) + c$ donne un résultat différent de $a \times (b + c)$. Avec nos équations logiques, on peut trouver des situations similaires : par exemple, $(a.b) + c$ est différent de $a.(b + c)$. On est alors obligé d'utiliser des parenthèses.

Méthode des *Minterms*

Reste à savoir comment transformer une table de vérité en équations logiques, et enfin en circuit. Pour cela, il n'y a pas trente-six solutions : on va écrire une équation logique qui permettra de calculer la valeur (0 ou 1) d'une sortie en fonction de toutes les entrées du circuits. Et on fera cela pour toutes les sorties du circuit que l'on veut concevoir.

Pour cela, on peut utiliser ce qu'on appelle la méthode des ***minterms***. Cette méthode permet de découper un circuit en quelques étapes simples :

- lister les lignes de la table de vérité pour lesquelles la sortie vaut 1 ;
- écrire l'équation logique pour chacune de ces lignes ;
- faire un ***OU*** entre toutes ces équations logiques, en n'oubliant pas de les entourer par des parenthèses

Il ne reste plus qu'à faire cela pour toutes les sorties du circuit, et le tour est joué. Pour illustrer le tout, on va reprendre notre exemple avec le bit de parité.

Première étape

La première étape consiste donc à **lister les lignes de la table de vérité dont la sortie est à 1**.

Entrée e2	Entrée e1	Entrée e0		Sortie s0
0	0	0		0
0	0	1		1
0	1	0		1
0	1	1		0
1	0	0		1
1	0	1		0
1	1	0		0
1	1	1		1

Deuxième étape

Ensuite, on doit **écrire l'équation logique de chacune des lignes sélectionnées à l'étape d'avant**.

Pour écrire l'équation logique d'une ligne, il faut simplement :

- lister toutes les entrées de la ligne ;
- faire un ***NON*** sur chaque entrée à 0 ;
- et faire un ***ET*** avec le tout.

Par exemple, prenons la première ligne dont la sortie vaut 1, à savoir la deuxième.

Entrée e2	Entrée e1	Entrée e0
0	0	1

L'équation logique de cette ligne sera donc : $\overline{e2}.\overline{e1}.e0$.

Il faut ensuite faire cela pour toutes les lignes dont la sortie vaut 1.

Seconde ligne :

Entrée e2	Entrée e1	Entrée e0
0	1	0

L'équation logique de cette ligne sera donc : $\overline{e2}..e1\overline{e0}$.

Troisième ligne :

Entrée e2	Entrée e1	Entrée e0
1	0	0

L'équation logique de cette ligne sera donc : $e2.\overline{e1}.\overline{e0}$.

Quatrième ligne :

Entrée e2	Entrée e1	Entrée e0
1	1	1

L'équation logique de cette ligne sera donc : $e2.e1.e0$.

Troisième étape

On a alors obtenu nos équations logiques. Reste à faire un bon gros ***OU*** entre toutes ces équations, et le tour est joué ! On obtient alors l'équation logique suivante : $(\overline{e2}.\overline{e1}.e0) + (\overline{e2}.e1.\overline{e0}) + (e2.\overline{e1}.\overline{e0}) + (e2.e1.e0)$

A ce stade, vous pourriez traduire cette équation directement en circuit, mais il y a un petit inconvénient...

Simplifications du circuit

Comme on l'a vu, on fini par obtenir une équation logique qui permet de décrire notre circuit. Mais quelle équation : on se retrouve avec un gros paquet de ***ET*** et de ***OU*** un peu partout ! Autant dire qu'il serait sympathique de pouvoir **simplifier** cette équation. Bien sûr, on peut vouloir simplifier cette équation juste pour se simplifier la vie lors de la traduction de cette équation en circuit, mais cela sert aussi à autre chose : cela permet d'obtenir un circuit plus rapide et/ou utilisant moins de portes logiques. Autant vous dire qu'apprendre à simplifier ces équations est quelque chose de crucial, particulièrement si vous voulez concevoir des circuits un tant soit peu rapides.

Pour donner un exemple, sachez que la grosse équation logique obtenue auparavant : $(\overline{e2}.\overline{e1}.e0) + (\overline{e2}.e1.\overline{e0}) + (e2.\overline{e1}.\overline{e0}) + (e2.e1.e0)$; peut se simplifier en : $e2 \oplus e1 \oplus e0$ avec les règles de simplifications vues au-dessus. Dans cet exemple, on passe donc de 17 portes logiques à seulement 3 !

Pour simplifier notre équation, on peut utiliser certaines propriétés mathématiques simples de ces équations. Ces propriétés forment ce qu'on appelle l'**algèbre de Boole**, du nom du mathématicien qui les a découvertes/inventées.

Règle	Description
Commutativité	$a + b = b + a$ $a.b = b.a$ $a \oplus b = b \oplus a$
Associativité	$(a + b) + c = a + (b + c)$ $(a.b).c = a.(b.c)$ $(a \oplus b) \oplus c = a \oplus (b \oplus c)$
Distributivité	$(a + b).c = (c.b) + (c.a)$ $(a.b) + c = (c + b).(c + a)$
Idempotence	$a.a = a$ $a + a = a$
Element nul	$a.0 = 0$ $a + 1 = 1$
Element Neutre	$a.1 = a$ $a + 0 = a$
Loi de De Morgan	$\overline{a} + \overline{b} = \overline{a.b}$; $\overline{a}.\overline{b} = \overline{a + b}$.
Complémentarité	$\overline{\overline{a}} = a$ $a + \overline{a} = 1$ $a.\overline{a} = 0$

On peut aussi rajouter que la porte ***XOR*** a ses propres règles.

Regle	Description

XOR	$a \oplus b = (\overline{a}.b) + (a.\overline{b})$
	$a \oplus 0 = a$
	$a \oplus 1 = \overline{a}$
	$a \oplus a = 0$
	$a \oplus \overline{a} = 1$

En utilisant ces règles algébriques, on peut arriver à simplifier une équation assez rapidement. On peut ainsi factoriser ou développer certaines expressions, comme on le ferait avec une équation normale, afin de simplifier notre équation logique. Le tout est de bien faire ces simplifications en appliquant correctement ces règles. Pour cela, il n'y a pas de recette miracle : vous devez sortir votre cerveau, et réfléchir !

Il existe d'autres méthodes pour simplifier nos circuits. Les plus connues étant les **tableaux de Karnaugh** et l'**algorithme de Quine Mc Cluskey**. On ne parlera pas de ces méthodes, qui sont assez complexes et n'apporteraient rien dans ce tutoriel. Il faut dire que ces méthodes risquent de ne pas vraiment nous servir : elles possèdent quelques défauts qui nous empêchent de créer de très gros circuits avec. Pour le dire franchement, elles sont trop longues à utiliser quand le nombre d'entrée du circuit dépasse 5 ou 6.

Mais

Un des problèmes des approches mentionnées plus haut est qu'elles nécessitent de créer une table de vérité. Et plus on a d'entrées, plus la table devient longue, et cela prend du temps pour la remplir. Cela ne pose aucun problèmes pour créer des circuits de moins de 5 ou 6 variables, mais au-delà, il y a de quoi rendre les armes assez rapidement. Et si vous ne me croyez pas, essayez de remplir la table de vérité d'un circuit qui additionne deux nombres de 32 bits, vous verrez : cela vous donnera une table de vérité de 4 294 967 296 lignes. Je ne sais pas si quelqu'un a déjà essayé de créer une telle table et d'en déduire le circuit correspondant, mais si c'est le cas, j'aurais de sérieuses craintes sur sa santé mentale. Pour compenser, on doit donc ruser.

Pour cela, il n'y a qu'une seule solution : on doit découper notre circuit en circuits plus petits qu'on relie ensemble. Il suffit de continuer ce découpage tant qu'on ne peut pas appliquer les techniques vues plus haut.

Circuits séquentiels

Avec le premier chapitre, on sait coder de l'information. Avec le second chapitre et la partie sur les circuits combinatoires, on sait traiter et manipuler de l'information. Il nous manque encore une chose : savoir comment faire pour mémoriser de l'information. Les circuits combinatoires n'ont malheureusement pas cette possibilité et ne peuvent pas stocker de l'information pour l'utiliser quand on en a besoin. La valeur de la sortie de ces circuits ne dépend que de l'entrée, et pas de ce qui s'est passé auparavant : les circuits combinatoires n'ont pas de **mémoire**. Ils ne peuvent qu'effectuer un traitement sur des données immédiatement disponibles. On n'irait pas loin en se contentant de ce genre de circuits : il serait totalement impossible de créer un ordinateur.

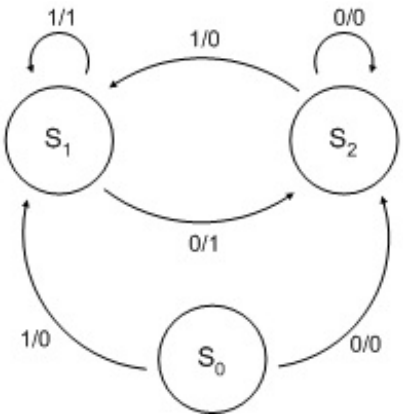
Comment donner de la mémoire à nos circuits ?

Mais rassurez-vous, tout n'est pas perdu ! Il existe des circuits qui possèdent une telle capacité de mémorisation : ce sont les **circuits séquentiels**. Ces circuits sont donc capables de mémoriser des informations, et peuvent les utiliser pour déterminer quoi mettre sur leurs sorties. L'ensemble de ces informations mémorisées dans notre circuit forme ce qu'on appelle l'**état** de notre circuit.

Pour mémoriser des informations (un état), notre circuit doit posséder des circuits spéciaux, chacun d'entre eux pouvant stocker un ou plusieurs bits, qu'on appelle des **mémoires**. On verra dans la suite de ce tutoriel comment les mémoires actuelles font pour stocker des bits : elles peuvent utiliser aussi bien un support magnétique (disques durs), optique (CD-ROM, DVD-ROM, etc), que des transistors (mémoires RAM, FLASH, ROM, etc), etc.

Reste que cet état peut changer au cours du fonctionnement de notre circuit. Rien n'empêche de vouloir modifier les informations mémorisées dans un circuit. On peut faire passer notre circuit séquentiel d'un état à un autre sans trop de problèmes. Ce passage d'un état à un autre s'appelle une **transition**.

Un circuit séquentiel peut être intégralement décrit par les états qu'il peut prendre, ainsi que par les transitions possibles entre états. Si vous voulez concevoir un circuit séquentiel, tout ce que vous avez à faire est de lister tous les états possibles, et quelles sont les transitions possibles. Pour ce faire, on utilise souvent une représentation graphique, dans laquelle on représente les états possibles du circuit par des cercles, et les transitions possibles par des flèches.



La transition effectuée entre deux états dépend souvent de ce qu'on met sur l'entrée du circuit. Aussi bien l'état du circuit (ce qu'il a mémorisé) que les valeurs présentes sur ses entrées, vont déterminer ce qu'on trouve sur la sortie. Par exemple, la valeur présente sur l'entrée peut servir à mettre à jour l'état ou donner un ordre au circuit pour lui dire : change d'état de tel ou tel façon. Dans la suite du tutoriel, vous verrez que certains composants de notre ordinateur fonctionnent sur ce principe : je pense notamment au processeur, qui contient des mémoires internes décrivant son état (des registres), et que l'on fait changer d'état via des instructions fournies en entrée.

Pour rendre possible les transitions, on doit mettre à jour l'état de notre circuit avec un circuit combinatoire qui décide quel sera le nouvel état de notre circuit en fonction de l'ancien état et des valeurs des entrées. Un circuit séquentiel peut donc (sans que ce soit une obligation) être découpé en deux morceaux : une ou plusieurs mémoires qui stockent l'état de notre circuit, et un ou plusieurs circuits combinatoires chargés de mettre à jour l'état du circuit, et éventuellement sa sortie.

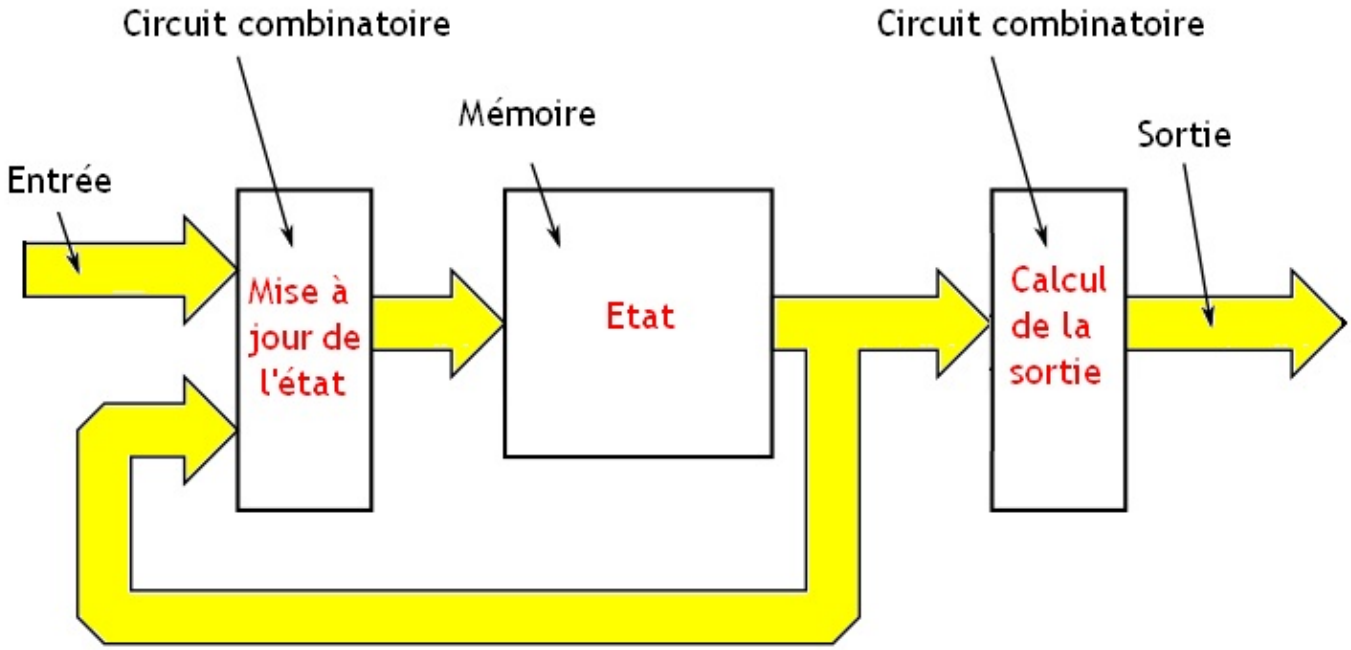
Pour la culture générale, il existe principalement deux types de circuits séquentiels :

- les automates de Moore ;
- et les Automates de Mealy.

Automates de Moore

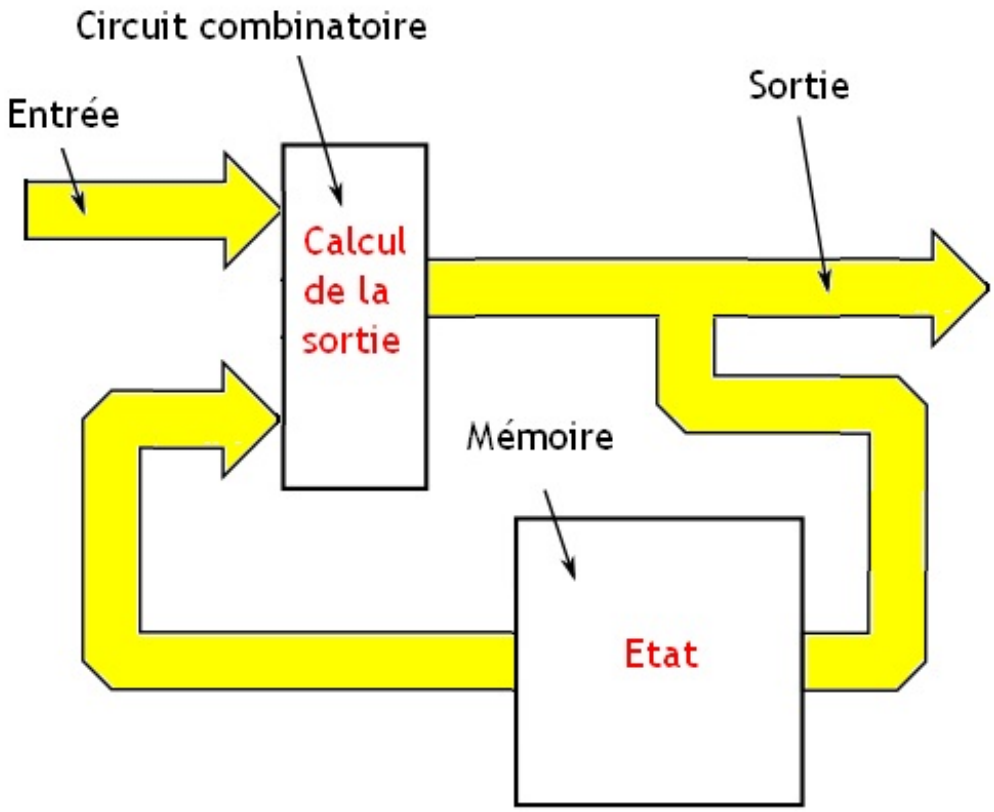
Avec les **automates de Moore**, ce qu'on trouve en sortie ne dépend que de l'état de l'automate. On peut donc simplement placer un circuit combinatoire qui se chargera de lire l'état de l'automate et qui fournira un résultat sur la sortie directement.

Pour mettre à jour l'état, on place un circuit combinatoire qui va prendre les entrées du circuit, ainsi que l'état actuel du circuit (fourni sur la sortie), et qui déduira le nouvel état, les nouvelles données à mémoriser.



Automates de Mealy

Autre forme de circuits séquentiels : les **automates de Mealy**. Avec ceux-ci, la sortie dépend non seulement de l'état du circuit, mais aussi de ce qu'on trouve sur les entrées.



Ces automates ont tendance à utiliser moins de portes logiques que les automates de Moore.

Bascules

On a vu plus haut que la logique séquentielle se base sur des circuits combinatoires, auxquels on a ajouté des mémoires. Pour le moment, on sait créer des circuits combinatoires, mais on ne sait pas faire des mémoires. Pourtant, on a déjà tout ce qu'il faut : avec nos portes logiques, on peut créer des circuits capables de mémoriser un bit. Ces circuits sont ce qu'on appelle des **bascules**.

En assemblant plusieurs de ces bascules ensembles, on peut créer ce qu'on appelle des **registres**, des espèces de mémoires assez rapides qu'on retrouve un peu partout dans nos ordinateurs : presque tous les circuits présents dans notre ordinateur contiennent des registres, que ce soit le processeur, la mémoire, les périphériques, etc.

Principe

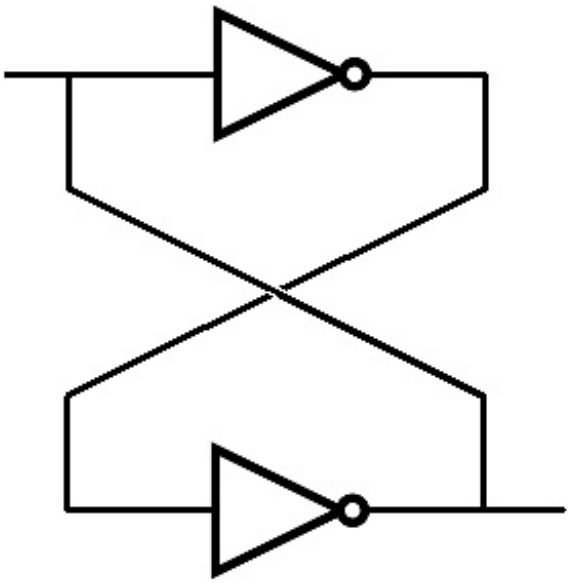
Une solution pour créer une bascule consiste à boucler la sortie d'un circuit sur son entrée, de façon à ce que la sortie rafraîchisse le contenu de l'entrée en permanence et que le tout forme une boucle qui s'auto-entretienne. Une bonne partie des circuits séquentiels contiennent des boucles quelque part, avec une entrée reliée sur une sortie. Ce qui est tout le contraire des circuits combinatoires, qui ne contiennent jamais la moindre boucle !

Bien sur, cela ne marche pas avec tous les circuits : dans certains cas, cela ne marche pas, ou du moins cela ne suffit pas pour mémoriser des informations. Par exemple, si je relie la sortie d'une porte ***NON*** à son entrée, le montage obtenu ne sera pas capable de mémoriser quoique ce soit.

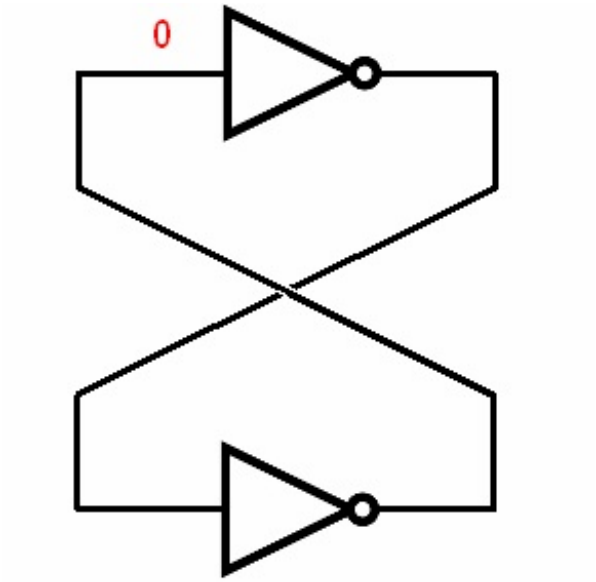
Et si on essayait avec deux portes ***NON*** ?

Ah, c'est plutôt bien vu !

En effet, en utilisant deux portes ***NON***, et en les reliant comme indiqué sur les schéma juste en dessous, on peut mémoriser un bit.

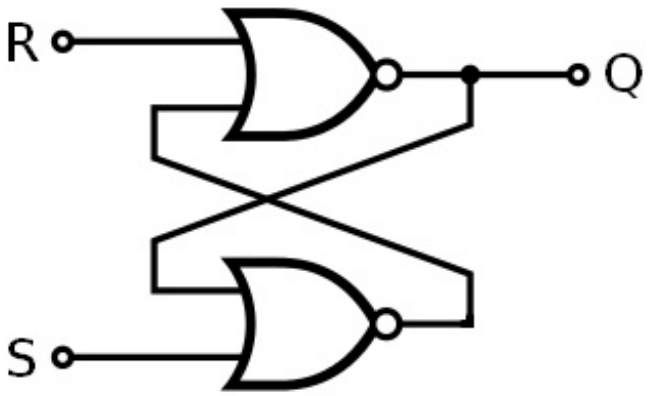


Si on place l'entrée de la première porte ***NON*** à zéro, la sortie de celle-ci passera à 1. Cette sortie sera reliée à l'entrée de l'autre porte ***NON***, qui inversera ce 1, donnant un zéro. Zéro qui sera alors ré-envoyé sur l'entrée initiale. L'ensemble sera stable : on peut déconnecter l'entrée du premier inverseur, celle-ci sera alors rafraichie en permanence par l'autre inverseur, avec sa valeur précédente. Le même raisonnement fonctionne si on met un 1 en sortie.

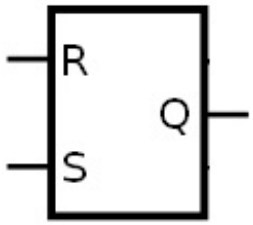


Bascule RS à NOR

Le seul problème, c'est qu'il faut bien mettre à jour l'état de ce bit de temps en temps. Il faut donc ruser. Pour mettre à jour l'état de notre circuit, on va simplement rajouter une entrée à notre circuit qui servira à le mettre à jour, et remplacer notre porte ***NON*** par une porte logique qui se comportera comme un inverseur dans certaines conditions. Le tout est de trouver une porte logique qui inverse le bit venant de l'autre inverseur si l'autre entrée est à zéro (ou à 1, suivant la bascule). Des portes ***NOR*** font très bien l'affaire.



On obtient alors ce qu'on appelle des **bascules RS**. Celles-ci sont des bascules qui comportent deux entrées R et S, et une sortie Q, sur laquelle on peut lire le bit stocké.



Le principe de ces bascules est assez simple :

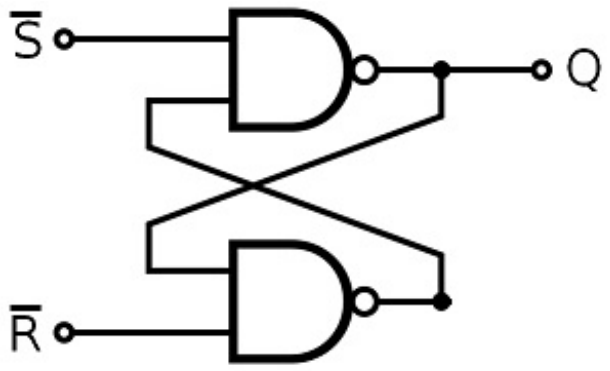
- si on met un 1 sur l'entrée R et un 0 sur l'entrée S, la bascule mémorise un zéro ;
- si on met un 0 sur l'entrée R et un 1 sur l'entrée S, la bascule mémorise un un ;
- si on met un zéro sur les deux entrées, la sortie Q sera égale à la valeur mémorisée juste avant.

Pour vous rappeler de ceci, sachez que les entrées de la bascule ne sont nommées ainsi par hasard : R signifie *Reset* (qui signifie mise à zéro en anglais), et S signifie *Set* (qui veut dire Mise à un en anglais). Petite remarque : si on met un 1 sur les deux entrées, le circuit ne répond plus de rien. On ne sait pas ce qui arrivera sur ses sorties. C'est bête, mais c'est comme ça !

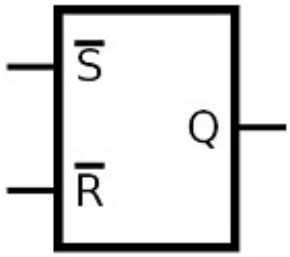
Entrée Reset	Entrée Set	Sortie Q
0	0	Bit mémorisé par la bascule
0	1	1
1	0	0
1	1	Interdit

Bascules RS à NAND

On peut aussi utiliser des portes ***NAND*** pour créer une bascule.



En utilisant des portes ***NAND***, le circuit change un peu. Celles-ci sont des bascules qui comportent deux entrées $\overline{R}$ et $\overline{S}$, et une sortie Q, sur laquelle on peut lire le bit stocké.



Ces bascules fonctionnent différemment de la bascule précédente :

- si on met un 0 sur l'entrée $\overline{R}$ et un 1 sur l'entrée $\overline{S}$, la bascule mémorise un 0 ;
- si on met un 1 sur l'entrée $\overline{R}$ et un 0 sur l'entrée $\overline{S}$, la bascule mémorise un 1 ;
- si on met un 1 sur les deux entrées, la sortie Q sera égale à la valeur mémorisée juste avant.

Entrée Reset	Entrée Set	Sortie Q
0	0	Interdit
0	1	0
1	0	1
1	1	Bit mémorisé par la bascule

Bascule D

Comme vous le voyez, notre bascule RS est un peu problématique : il y a une combinaison d'entrées pour laquelle on ne sait pas ce que va faire notre circuit. On va devoir résoudre ce léger défaut.

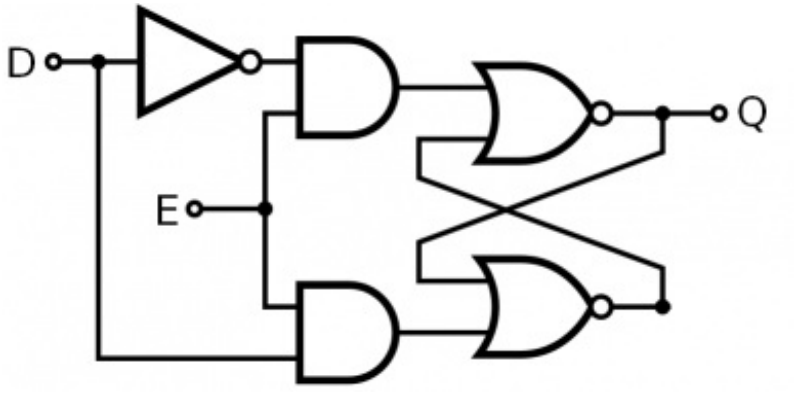
Tout d'abord, il faut remarquer que la configuration problématique survient quand on cherche à mettre R et S à 1 en même temps. Or, le bit R permet de mettre à zéro notre bascule, tandis que le bit S va la mettre à 1. Pas étonnant que cela ne marche pas. Pour résoudre ce problème, il suffit simplement de remarquer que le bit R est censé être l'exact opposé du bit S : quand on veut mettre un bit à 1, on ne le met pas zéro, et réciproquement. Donc, on peut se contenter d'un bit, et ajouter une porte ***NON*** pour obtenir l'autre bit.

Dans ce qui suit, on va choisir de garder le bit S. Pour une raison très simple : en faisant cela, placer un 0 sur l'entrée S fera mémoriser un zéro à la bascule, tandis qu'y placer un 1 mémorisera un 1. En clair, l'entrée S contiendra le bit à mémoriser.

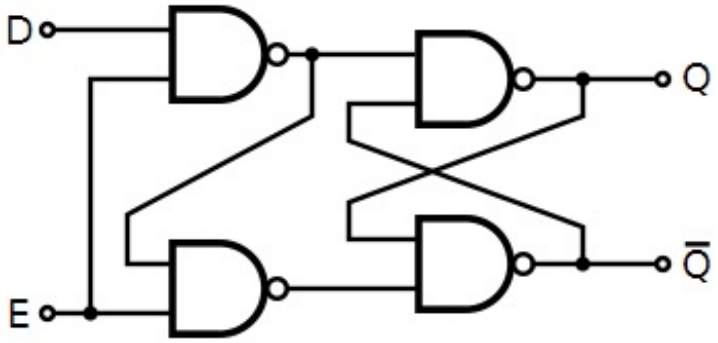
Image utilisateur

Mais, il y a un petit problème. Si on regarde la table de vérité de ce nouveau circuit, on s'aperçoit qu'il ne mémorise rien ! Si on place un 1 sur l'entrée R, la bascule sera mise à 1, et si on met un zéro, elle sera mise à zéro. Pour régler ce petit problème, on va rajouter une entrée, qui permettra de dire à notre bascule : ne prend pas en compte ce que tu trouve sur ton entrée S. Cette entrée, on va l'appeler l'entrée de validation d'écriture. Elle servira à autoriser l'écriture dans la bascule.

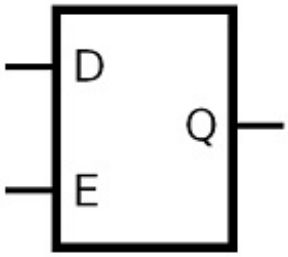
Reste à savoir quoi rajouter dans notre circuit pour ajouter cette entrée. En réfléchissant bien, on se souvient que notre bascule RS effectuait une mémorisation quand ses bits R et S étaient tous les deux à 0. Ce qu'il faut rajouter, ce sont des portes, reliées à ce qui était autrefois les entrées R et S, reliées à notre nouvelle entrée. Il suffit que ces portes envoient un zéro sur leur sortie quand l'entrée de validation d'écriture est à zéro, et recopie son autre entrée sur sa sortie dans le cas contraire. Ce qu'on vient de décrire est exactement le fonctionnement d'une porte ***ET***. On obtient alors le circuit suivant.



On peut aussi faire la même chose, mais avec la bascule RS à NAND.

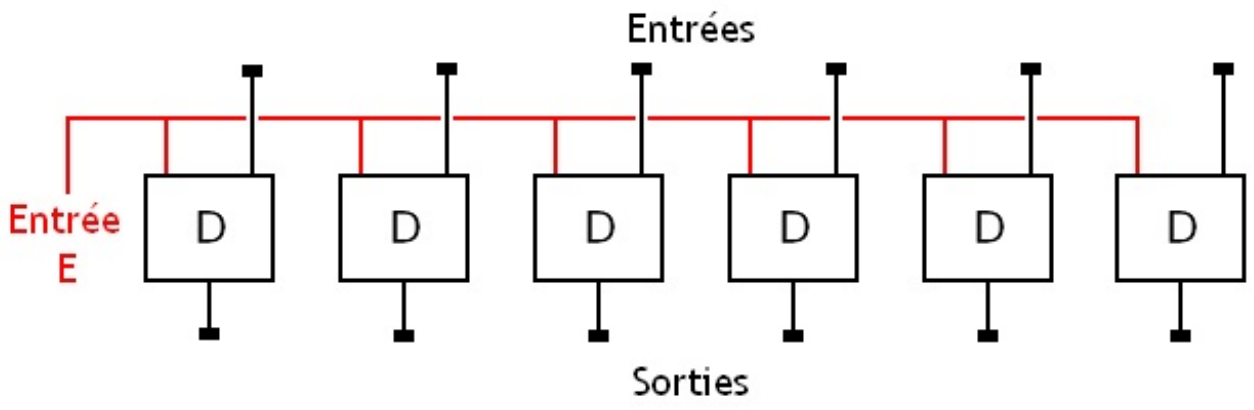


Ce qu'on vient de fabriquer s'appelle une **bascule D**.



Mémoires

A partir de ces petites mémoires de 1 bit, on peut créer des mémoires un peu plus conséquentes. Grâce à cela, on saura maintenant créer des circuits séquentiels ! Pour commencer, il faut remarquer que la mémoire d'un circuit séquentiel forme un tout : on ne peut pas en modifier un morceau : lors d'une transition, c'est toute la mémoire de l'automate qui est modifié. Donc, on doit faire en sorte que la mise de nos mémoire se fasse en même temps. Rien de plus simple : il suffit de prendre plusieurs bascules D pour créer notre mémoire, et de relier ensemble leurs entrées de validation d'écriture.



C'est ainsi que son créer les mémoires qui sont internes à nos circuits séquentiels. Vous verrez que beaucoup des circuits d'un ordinateur sont des circuits séquentiels, et que ceux-ci contiennent toujours des petites mémoires, fabriquées à l'aide de bascules. Ces petites mémoires, que l'on vient de créer, sont appelées des **registres**.

Tic, Tac, Tic, Tac : Le signal d'horloge

Visiblement, il ne manque rien : on sait fabriquer des mémoires et des circuits combinatoires, rien ne peut nous arrêter dans notre marche vers la conception d'un ordinateur. Tout semble aller pour le mieux dans le meilleur des mondes. Sauf que non, on a oublié de parler d'un léger détail à propos de nos circuits.

Temps de propagation

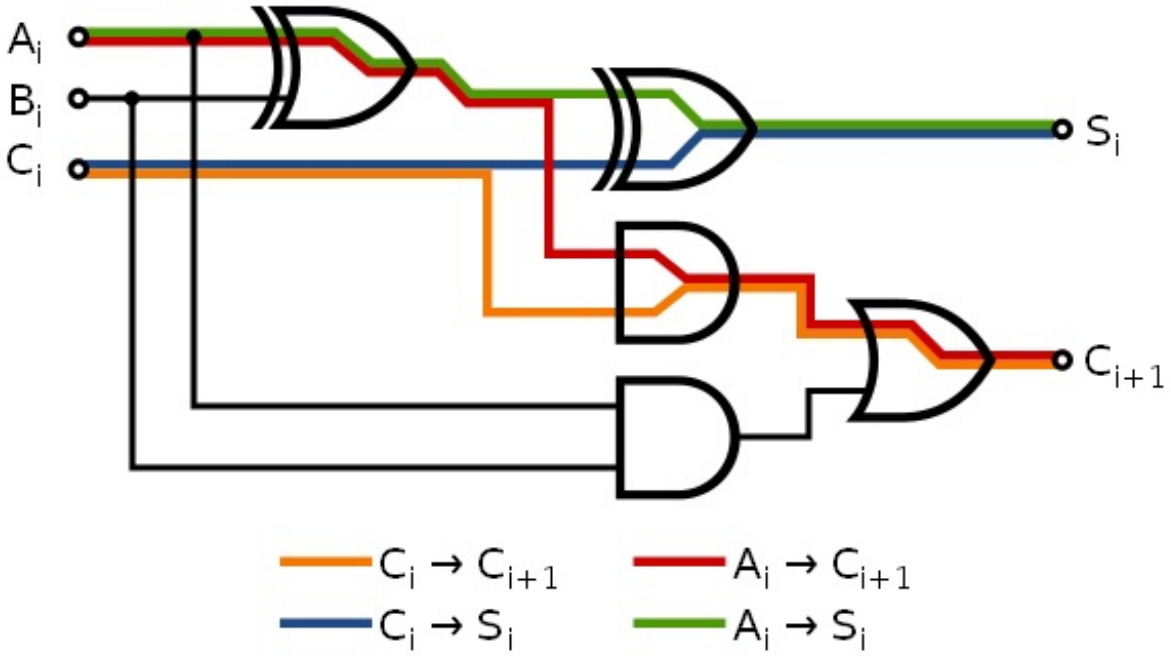
Tout circuit, quel qu'il soit, va mettre un petit peu de temps avant de réagir. Ce temps mit par le circuit pour s'apercevoir qu'il s'est passé quelque chose sur son entrée et modifier sa sortie en conséquence s'appelle le **temps de propagation**. Pour faire simple, c'est le temps que met un circuit à faire ce qu'on lui demande. Pour en donner une définition plus complète, on peut dire que c'est le temps entre le moment pendant lequel on modifie la tension sur une entrée d'un circuit logique et le moment où cette modification se répercute sur les sorties.

Ce temps de propagation dépend fortement du circuit et peut dépendre de pas mal de paramètres. Mais il y a trois raisons principales, qui sont à l'origine de ce temps de propagation. Il va de soit que plus ce temps de propagation est élevé, plus notre circuit risque d'être lent, et savoir sur quoi jouer pour le diminuer n'est pas un luxe. Voyons donc ce qu'il en est.

Critical Path

Le plus important de ces paramètres est ce qu'on appelle le ***Critical Path*** . Il s'agit du nombre maximal de portes logiques entre une entrée et une sortie de notre circuit.

Pour donner un exemple, nous allons prendre le schéma suivant.



Pour ce circuit, le *Critical Path* est le chemin dessiné en rouge. En suivant ce chemin, on va traverser 3 portes logiques, contre deux ou une dans les autres chemins. Pour information, tous les chemins possibles ne sont pas présentés sur le schéma, mais ceux qui ne sont pas représentés passent par moins de 3 portes logiques.

De plus, on doit préciser que nos portes n'ont pas toute le même temps de propagation : une porte ***NON*** aura tendance à être plus rapide qu'une porte ***NAND***, par exemple.

Fan Out

Autre facteur qui joue beaucoup sur ce temps de propagation : le nombre de composants reliés sur la sortie d'une porte logique. Plus on connecte de portes logiques sur un fil, plus il faudra du temps pour la tension à l'entrée de ces portes change pour atteindre sa bonne valeur.

Wire Delay

Autre facteur qui joue dans le temps de propagation : le temps mis par notre tension pour se propager dans les "fils" et les interconnexions qui relient les portes logiques entre elles. Ce temps dépend notamment de la résistance (celle de la loi d'Ohm, que vous avez surement déjà vue il y a un moment) et de ce qu'on appelle la capacité des interconnexions. Ce temps perdu dans les fils devient de plus en plus important au fil du temps, les transistors et portes logiques devenant de plus en plus rapides à force des les miniaturiser. Pour donner un exemple, sachez que si vous comptez créer des circuits travaillant sur des entrées de 256 à 512 bits qui soient rapides, il vaut mieux modifier votre circuit de façon à minimiser le temps perdu dans les interconnexions au lieu de diminuer le *Critical Path*.

Circuits synchrones

Ce temps de propagation doit être pris en compte quand on crée un circuit séquentiel. Sans cela on ne sait pas quand mettre à jour la mémoire intégrée dans notre circuit séquentiel. Si on le fait trop tôt, le circuit ne se comportera pas comme il faut : on peut parfaitement sauter des états. De plus, les différents circuits d'un ordinateur n'ont pas tous le même temps de propagation, et ceux-ci vont fonctionner à des vitesses différentes. Si l'on ne fait rien, on peut se retrouver avec des dysfonctionnements : par exemple, un composant lent peut donc rater deux ou trois ordres successifs envoyées par un composant un peu trop rapide.

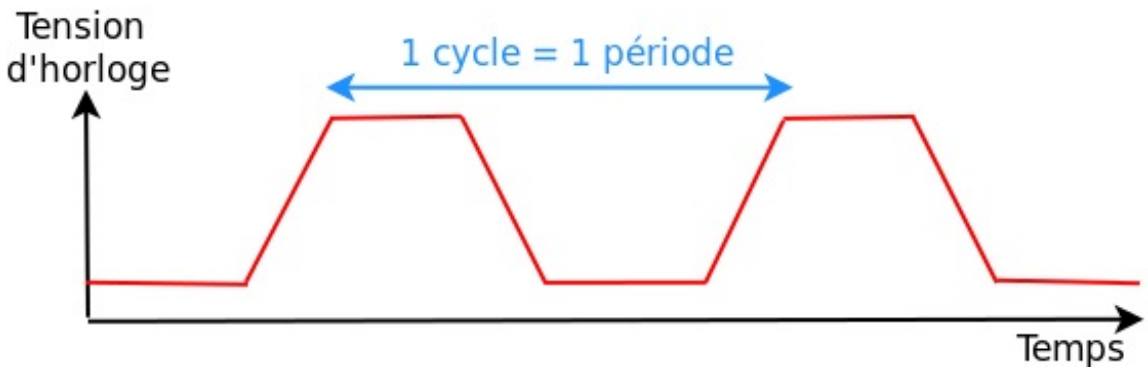
Comment éviter les ennuis dus à l'existence de ce temps de propagation ?

Il existe diverses solutions. On peut notamment faire en sorte que les entrées et le circuit combinatoire prévienne la mémoire quand ils veulent la mettre à jour. Quand l'entrée et le circuit combinatoire sont prêts, on autorise l'écriture dans la mémoire. C'est ce qui est fait dans les **circuits asynchrones**. Mais ce n'est pas cette solution qui est utilisée dans nos ordinateur.

La majorité des circuits de nos ordinateur gèrent les temps de propagation différemment. Ce sont ce qu'on appelle des **circuits synchrones**. Pour simplifier, ces circuits vont mettre à jour leurs mémoires à intervalles réguliers. La durée entre deux mises à jour est constante et doit être plus grande que le pire temps de propagation possible du circuit. Les concepteurs d'un circuit doivent estimer le pire temps de propagation possible pour le circuit et ajouter une marge de sureté.

L'horloge

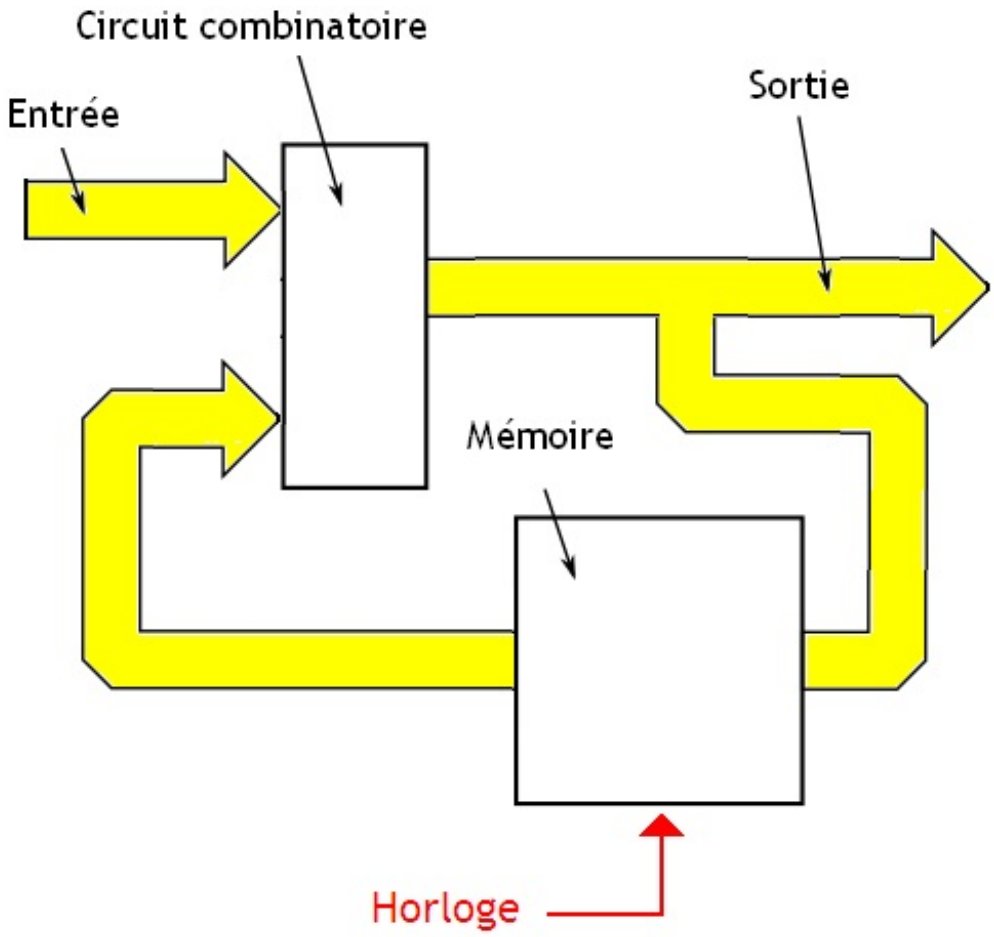
Pour mettre à jour nos circuits à intervalles réguliers, ceux-ci sont commandés par une tension qui varie de façon cyclique : le **signal d'horloge**. Celle-ci passe de façon cyclique de 1 à 0. Cette tension effectue un cycle plusieurs fois par seconde. Le temps que met la tension pour effectuer un cycle est ce qu'on appelle la **période**. Le nombre de cycle, de périodes, en une seconde est appelé la **fréquence**. Cette fréquence se mesure dans une unité : le hertz.



On voit sur ce schéma que la tension ne peut pas varier instantanément : la tension met un certain temps pour passer de 0 à 1 et de 1 à 0. On appelle cela un **front**. La passage de 0 à 1 est appelé un **front montant** et le passage de 1 à 0 un **front descendant**.

Les circuits

Cette horloge est reliée aux entrées d'autorisation d'écriture des bascules du circuit. Pour cela, on doit rajouter une entrée sur notre circuit, sur laquelle on enverra l'horloge.



En faisant cela, notre circuit logique va "lire" les entrées et en déduire une sortie uniquement lorsqu'il voit un front montant (ou descendant) sur son entrée d'horloge ! Entre deux fronts montants (ou descendants), notre circuit est complétement autiste du point de vue des entrées : on peut faire varier autant de fois qu'on veut nos entrées, il faudra attendre le prochain front montant pour notre circuit réagisse.

Dans le cas où notre circuit est composé de plusieurs sous-circuits devant être synchronisés via l'horloge, celle-ci est distribuée à tous les sous-circuits à travers un réseau de connections électriques qu'on appelle l'**arbre d'horloge**.

Et dans nos PC ?

Dans la pratique, une bonne partie des composants d'un ordinateur sont synchronisés par des horloges. Oui, j'ai bien dit DES horloges. Par exemple, notre processeur fonctionne avec une horloge différente de l'horloge de la mémoire ! La présence de plusieurs horloges est justifiée par un fait très simple : certains composants informatiques sont plus lents que d'autres et ne sont pas capables de fonctionner avec des horloges rapides. Par exemple, le processeur a souvent une horloge très rapide comparée à l'horloge des autres composants. Généralement, plus un composant utilise une fréquence élevée, plus celui-ci est rapide. Cela n'est toutefois pas un élément déterminant : un processeur de 4 gigahertz peut être bien plus rapide qu'un processeur de 200 gigahertz, pour des raisons techniques qu'on verra plus tard dans ce tutoriel. De nos jours, c'est plus la façon dont notre processeur va faire ses opérations qui sera déterminante : ne vous faites pas avoir par le Megahertz Myth !

En fait, il existe une horloge de base qui est "transformée" en plusieurs horloges dans notre ordinateur. On peut parfaitement transformer un signal d'horloge en un autre, ayant une période deux fois plus grande ou plus petite, grâce à des montages électroniques spécialisés. Cela peut se faire avec des composants appelés des PLL ou encore avec des montages à portes logiques un peu particuliers, qu'on n'abordera pas ici.

Les premiers processeurs avaient une fréquence assez faible et étaient peu rapides. Au fil du temps, avec l'amélioration des méthodes de conception des processeurs, la fréquence de ceux-ci a commencée a augmenter. Ces processeurs sont devenus plus rapides, plus efficaces. Pour donner un ordre de grandeur, le premier microprocesseur avait une fréquence de 740 kilohertz (740 000 hertz). De nos jours, les processeurs peuvent monter jusqu'à plusieurs gigahertz : plusieurs milliards de fronts par secondes ! Quoiqu'il en soit, cette montée en fréquence est aujourd'hui terminée : de nos jours, les concepteurs de processeurs sont face à un mur et ne peuvent plus trop augmenter la fréquence de nos processeurs aussi simplement qu'avant.

Et pourquoi les concepteurs de processeurs ont-ils arrêtés d'augmenter la fréquence de nos processeurs ?

Augmenter la fréquence a tendance à vraiment faire chauffer le processeur plus que de raison : difficile de monter en fréquence dans ces conditions. Une grande part de cette dissipation thermique a lieu dans l'arbre d'horloge : environ 20% à 35%. Cela vient du fait que les composants reliés à l'arbre horloge doivent continuer à changer d'état tant que l'horloge est présente, et ce même quand ils sont inutilisés. C'est la première limite à la montée en puissance : la dissipation thermique est tellement importante qu'elle limite grandement les améliorations possibles et la montée en fréquence de nos processeurs.

Auparavant, un processeur était refroidi par un simple radiateur. Aujourd'hui, on est obligé d'utiliser un radiateur et un ventilateur, avec une pâte thermique de qualité tellement nos processeurs chauffent. Pour limiter la catastrophe, tous les fabricants de CPU cherchent au maximum à diminuer la température de nos processeurs. Pour cela, ils ont inventé diverses techniques permettant de diminuer la consommation énergétique et la dissipation thermique d'un processeur. Mais ces techniques ne suffisent plus désormais. C'est ce qui est appelé le ***Heat Wall***.
Et voilà, avec un cerveau en parfait état de marche, et beaucoup de temps devant vous, vous pouvez construire n'importe quel circuit imaginable et fabriquer un ordinateur. Du moins, en théorie : n'essayez pas chez vous. Bon, blague à part, avec ce chapitre, vous avez tout de même le niveau pour créer certains circuits présents dans notre ordinateur comme l'ALU. Sympa, non ?

Partie 2 : Architecture de base

Dans ce chapitre, on essaye de comprendre comment est organisé l'intérieur d'un ordinateur. On verra sa structure de base et nous parlerons des différents composants d'un PC : processeurs, RAM, périphériques, etc.

C'est quoi un ordinateur ?

Nos ordinateurs servent à manipuler de l'**information**. Cette information peut être une température, une image, un signal sonore, etc. Bref, du moment que ça se mesure, c'est de l'information. Cette information, ils vont devoir la transformer en quelque chose d'exploitable et de facilement manipulable, que ce soit aussi bien du son, que de la vidéo, du texte et pleins d'autres choses. Pour cela, on va utiliser l'astuce vue au chapitre précédent : on code chaque information grâce à un nombre.

Une fois l'information codée correctement sous la forme de nombres, il suffira d'utiliser une **machine à calculer** pour pouvoir effectuer des manipulations sur ces nombres, et donc sur l'information codée : une simple machine à calculer devient alors une machine à traiter de l'information. Un **ordinateur** est donc un calculateur. Mais par contre, tout calculateur n'est pas un ordinateur : par exemple, certains calculateurs ne comptent même pas en binaire. Mais alors, qu'est-ce qu'un ordinateur ?

Numérique versus analogique

Pour pouvoir traiter de l'information, la première étape est d'abord de coder celle-ci. On a vu dans le chapitre sur le binaire comment représenter des informations simples en utilisant le binaire. Mais ce codage, cette transformation d'information en nombre, peut être fait de plusieurs façons différentes, et coder des informations en binaire n'est pas le seul moyen.

Analogique versus numérique

Dans les grandes lignes, on peut identifier deux grands types de codage :

- le codage analogique ;
- et le codage numérique.

Le codage analogique	Celui-ci utilise des nombres réels : il code l'information avec des grandeurs physiques (des trucs qu'on peut mesurer par un nombre) comprises dans un intervalle. Un codage analogique a une précision théoriquement infinie : on peut par exemple utiliser toutes les valeurs entre 0 et 5 pour coder une information. Celle-ci peut alors prendre une valeur comme 1 , 2.2345646, ou pire...
Le codage numérique	Celui-ci utilise uniquement des suites de symboles (qu'on peut assimiler à des chiffres), assimilables à des nombres entiers pour coder les informations. Pour simplifier, le codage numérique va coder des informations en utilisant des nombres entiers codés dans une base, qui peut être 2, 3, 4, 16, etc. Les fameux symboles dont je viens de parler sont simplement les chiffres de cette base. Le codage numérique n'utilise qu'un nombre fini de valeurs, contrairement au codage analogique. Un code numérique a une précision figée et ne pourra pas prendre un grand nombre de valeurs (comparé à l'infini). Cela donnera des valeurs du style : 0, 0.12 , 0.24 , 0.36, 0.48... jusqu'à 2 volts.

Un **calculateur analogique** peut donc faire des calculs avec une précision très fine, et peut même faire certains calculs avec une précision impossible à atteindre avec un calculateur numérique : des dérivées, des intégrations, etc. Un calculateur numérique peut bien sûr effectuer des intégrations et dérivations, mais ne donnera jamais un résultat exact et se contentera de donner une approximation du résultat. Un calculateur analogique pourra donner un résultat exact, du moins en théorie : un calculateur analogique insensible aux perturbations extérieures et n'ayant aucune imperfection n'a pas encore été inventé.

Pour les **calculateurs numériques**, les nombres manipulés sont codés par des suites de symboles (des "chiffres", si vous préférez), et un calculateur numérique ne fera que transformer des suites de symboles en d'autres suites de symboles. Vous pouvez par exemple identifier chacun de ces symboles en un chiffre dans une base entière quelconque (pas forcément la base 10 ou 2).

Dans un ordinateur, les symboles utilisés ne peuvent prendre que deux valeurs : 0 ou 1. De tels symboles ne sont rien d'autre que les fameux bits du chapitre précédent, ce qui fait que notre ordinateur ne manipule donc que des bits : vous comprenez maintenant l'utilité du premier chapitre.

L'immunité au bruit

Vu ce qui a été dit précédemment, nos calculateurs numériques ne semblent pas vraiment très intéressants. Et pourtant, la grande majorité de nos composants et appareils électroniques (dont nos ordinateurs) sont des machines numériques ! C'est du au fait que les calculateurs analogiques ont un gros problème : ils ont une faible **immunité au bruit**.

Explication : un signal analogique peut facilement subir des perturbations qui vont changer sa valeur, de façon parfois assez drastique. Autant vous dire que si une de ces perturbations un peu violente arrive, le résultat qui arrive en sortie n'est vraiment pas celui attendu. Si un système est peu sensible à ces perturbations, on dit qu'il a une meilleure immunité au bruit.

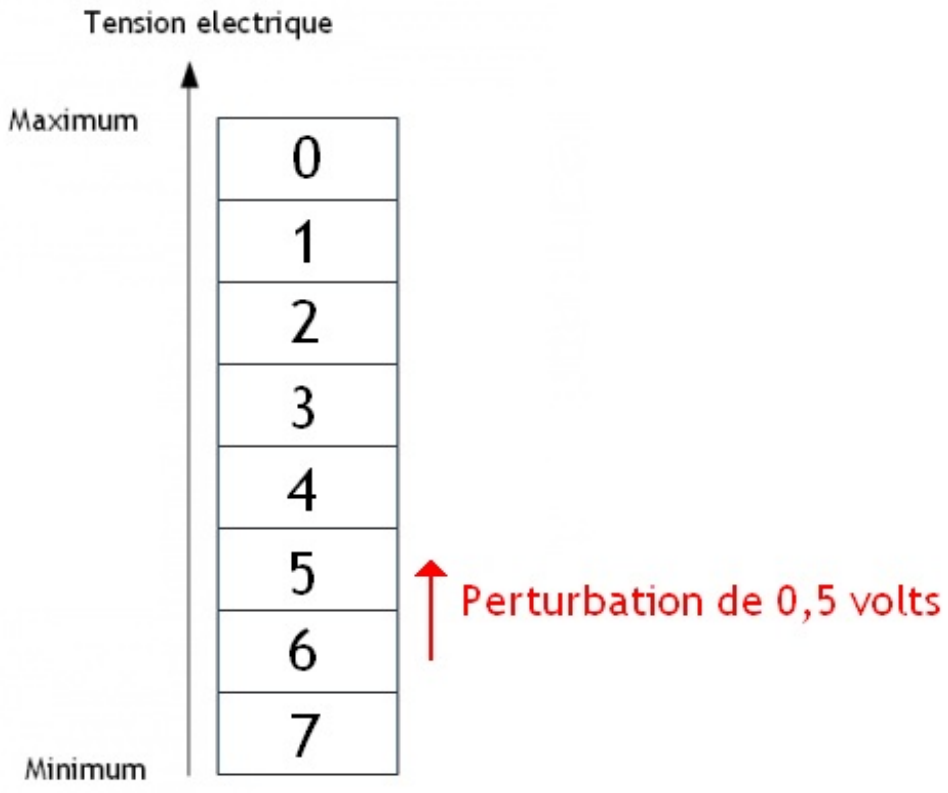
Un signal numérique n'a pas trop ce problème : les perturbations ou parasites vont moins perturber le signal numérique et vont éviter de trop modifier le signal original : l'erreur sera beaucoup plus faible qu'avec un signal analogique.

Mais pourquoi avoir choisi la base 2 dans nos ordinateurs ?

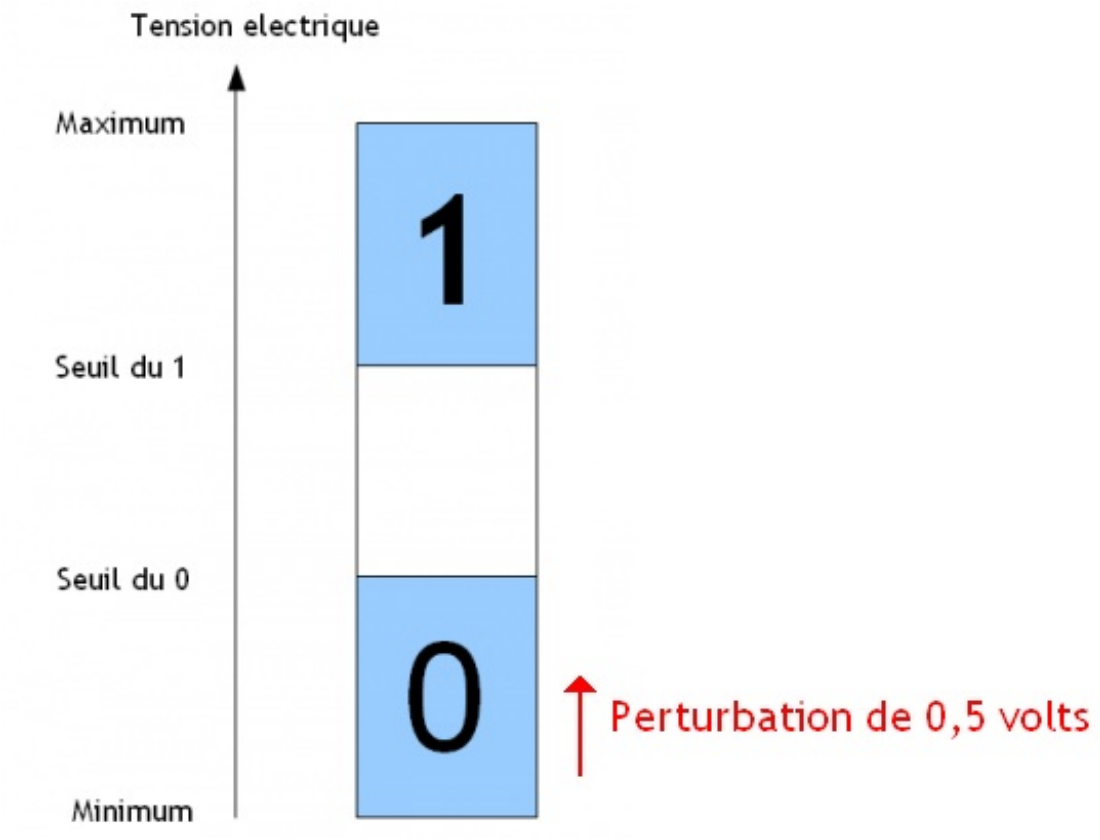
La question est parfaitement légitime : on aurait tout aussi bien pu prendre la base 10 ou n'importe quelle autre base. Il aurait été bien plus facile pour les humains qui doivent programmer ces machines d'utiliser la base 10. D'ailleurs, il existe de nombreuses machines qui manipulent des données numériques en base 10, en base 3, etc. Et on a déjà inventé des ordinateurs qui comptaient en base 3 : l'ordinateur SETUN, par exemple, fabriqué et conçu pour l'université de Moscou. Et rien n'empêche de créer des ordinateurs qui compteraient en base 10, 16, ou tout autre base. Mais il y a plusieurs raisons qui font que le binaire a été choisi comme base pour le codage de l'information dans un ordinateur.

La plus importante de toutes, c'est qu'une perturbation n'aura pas le même effet sur un nombre codé en base 2 et sur un nombre codé en base 10.

En effet, supposons que nous utilisions, par exemple, une tension comprise entre 0 et 9 volts, qui code un chiffre/symbole allant de 0 à 9 (on utilise donc la base 10). Le moindre parasite peut changer la valeur du chiffre codé par cette tension.



Avec cette tension qui code seulement un 0 ou un 1 (de 0volts pour un 0 et 10 pour un 1), un parasite de 1 volt aura nettement moins de chance de modifier la valeur du bit codé ainsi.



Le parasite aura donc un effet nettement plus faible : la résistance aux perturbations électromagnétiques extérieure est meilleure.

Architecture de base

Une fois notre information codée, il faut ensuite pouvoir la manipuler et la stocker. Ce traitement de notre information peut être fait de différentes façons. Pour transformer cette information et en faire quelque chose, il va falloir effectuer une série d'étapes. La première étape, c'est de coder cette information sous une forme utilisable. Mais ça ne fait pas tout, il faut encore traiter cette information.

I/O et traitement

Pour cela, on va donc devoir :

- recevoir une information codée par un nombre,
- modifier ce nombre et effectuer des calculs avec (de façon à en faire quelque chose),
- envoyer le résultat sur une sortie pour l'exploiter.

Toute machine traitant de l'information est donc composéé par :

- Une **entrée** sur laquelle on envoie une information.
- Une **unité de traitement**, qui va manipuler l'information une fois codée et donner un résultat codé sous la forme d'une suite de symboles ou d'un nombre.
- Une **sortie**, qui va prendre le résultat et en faire quelque chose (écrire sur une imprimante ou sur un moniteur, émettre du son,...).

Notre ordinateur contient pas mal d'entrées et de sorties. Par exemple, votre écran est une sortie : il reçoit des informations, et les

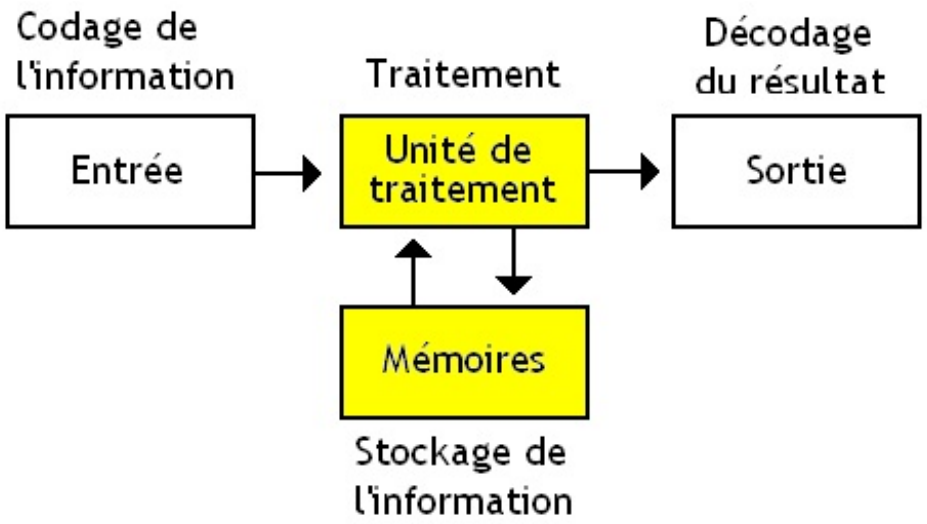
transforme en image affichée à l'écran. On pourrait aussi citer des dispositifs comme des imprimantes, ou des haut-parleurs. Comme entrée, vous avez votre clavier, votre souris, pour ne citer qu'eux.

Automates

Cette unité de traitement peut très bien consister en un vulgaire circuit combinatoire, ou tout autre mécanisme, sans mémoire. Mais d'autres unités de traitement ont une certaine capacité de mémorisation, comme les circuits séquentiels. Tout système dont l'unité de traitement possède cette capacité de mémorisation, et fonctionne comme un circuit électronique séquentiel, est appelé un **automate**.

Principe

Celui-ci contiendra donc des mémoires internes en plus de l'unité de traitement, qui représenteront l'état de l'automate (les informations qu'il a mémorisées). Bien sûr, cet état peut être mis à jour, et on peut changer l'état de notre automate pour modifier ces informations, les manipuler, etc. Notre unité de traitement pourra donc manipuler directement le contenu de nos mémoires. Notre automate passera donc d'états en états, via des suites d'étapes qui transformeront un état en un autre : on modifiera les informations contenues dans notre automate étapes par étapes jusqu'à arriver au résultat voulu. Ces changements d'état sont bien sur gouvernés par l'unité de traitement.



Attention : ce schéma est un schéma de principe. Il existe des automates pour lesquels il n'y a pas de séparation nette entre mémoire et circuits de traitement. Il est possible de créer des circuits dans lesquels la mémorisation des informations est entremêlée avec les circuits chargés de traiter l'information. Toutefois, dans nos ordinateurs, les deux sont relativement bien séparés, même si ce n'est pas totalement le cas. D'ailleurs, nos ordinateurs sont des automates spéciaux, composés à partir de composants plus petits qui sont eux-même des automates.

Pour information, on peut très bien créer des automates avec un peu n'importe quoi. Du genre, des dispositifs hydrauliques, ou électriques, magnétiques, voire à air comprimé. Pour citer un exemple, on peut citer le calculateur hydraulique MONIAC. Quant à nos ordinateurs, ils sont fabriqués avec des dispositifs électroniques, comme des portes logiques ou des montages à base de transistors et de condensateurs. Évidement, il existe des automates numériques, et des automates analogiques, voire des automates hybrides mélangeant des circuits analogiques et des circuits numériques.

Automate numérique

Dans un automate numérique (un ordinateur par exemple), l'information placée sur l'entrée est codée sous la forme d'une suite de symboles avant d'être envoyée à l'unité de traitement ou en mémoire. Nos informations seront codées par des suites de symboles, des nombres codés dans une certaine base, et seront stockées ainsi en mémoire. Les suites de symboles manipulées sont appelées des **données**. Dans nos ordinateurs, les symboles utilisés étant des zéros et des uns, nos données sont donc de simples suites de bits.

Reste que ces données seront manipulées par notre automate, par son unité de traitement. Tout ce que peut faire la partie traitement d'un automate numérique, c'est modifier l'état de l'automate, à savoir modifier le contenu des mémoires de l'automate. Cela peut permettre de transformer une (ou plusieurs) donnée en une (ou plusieurs) autre(s), ou de configurer l'automate pour qu'il fonctionne correctement. Ces transformations élémentaires qui modifient l'état de l'automate sont appelées des **instructions**. Un automate numérique est donc une machine qui va simplement appliquer une suite d'instructions dans un ordre bien précis sur les données. C'est cette suite d'instructions qui va définir le traitement fait par notre automate numérique, et donc ce à quoi il

peut servir.

Programme

Dans certains automates, la suite d'instructions effectuée est toujours la même. Une fois conçus, ceux-ci ne peuvent faire que ce pourquoi ils ont été conçus. Ils ne sont pas programmables. C'est notamment le cas pour les calculateurs analogiques : une fois qu'on a câblé un automate analogique, il est impossible de lui faire faire autre chose que ce pour quoi il a été conçu sans modifier son câblage. A la rigueur, on peut le reconfigurer et faire varier certains paramètres via des interrupteurs ou des boutons, mais cela s'arrête là. D'autres automates numériques ont le même problème : la suite d'instruction qu'ils exécutent est impossible à changer sans modifier les circuits de l'automate lui-même. Et cela pose un problème : à chaque problème qu'on veut résoudre en utilisant un automate, on doit recréer un nouvel automate. Autant dire que ça peut devenir assez embêtant !

Mais il existe une solution : créer des automates dont on peut remplacer la suite d'instructions qu'ils effectuent par une autre sans avoir à modifier leur câblage. On peut donc faire ce que l'on veut de ces automates : ceux-ci sont réutilisables à volonté et il est possible de modifier leur fonction du jour au lendemain et leur faire faire un traitement différent. On dit qu'ils sont **programmables**. Ainsi, pour programmer notre ordinateur, il suffira de créer une suite d'instructions qui va faire ce que l'on souhaite. Et c'est bien plus rapide que de créer un automate complet de zéro. Cette suite d'instruction sera alors appelée le **programme** de l'automate.

La solution utilisée pour rendre nos automates programmables consiste à stocker le programme dans une mémoire, qui sera modifiable à loisir. C'est ainsi que notre ordinateur est rendu programmable : on peut parfaitement modifier le contenu de cette mémoire (ou la changer, au pire), et donc changer le programme exécuté par notre ordinateur sans trop de problèmes. Mine de rien, cette idée d'automate stockant son programme en mémoire est ce qui a fait que l'informatique est ce qu'elle est aujourd'hui. C'est la définition même d'ordinateur : **automate programmable qui stocke son programme dans sa mémoire**.

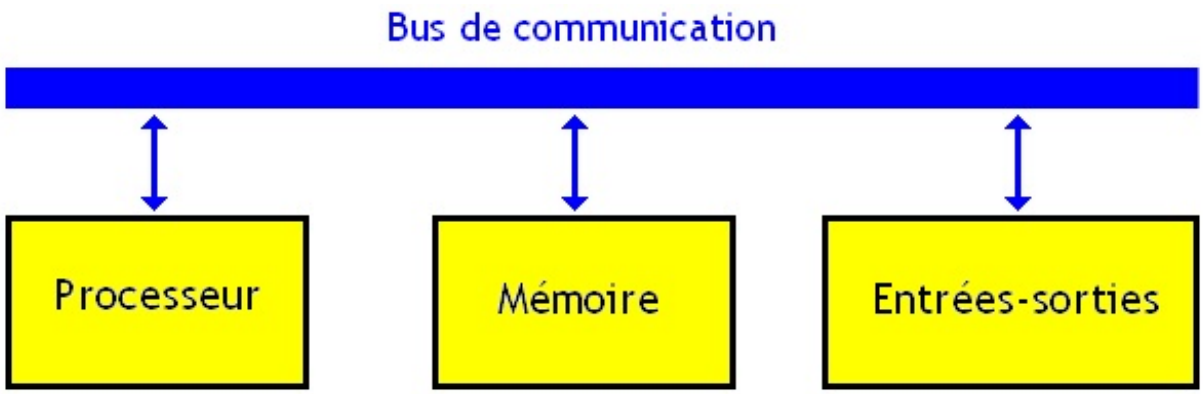
Ordinateurs

Tous nos ordinateurs sont plus ou moins organisés sur un même modèle de base, une organisation commune. Notre ordinateur est ainsi découpé en composants bien distincts, qui ont chacun une utilité particulière. Dans ce découpage en composant, on retrouve plus ou moins l'organisation qu'on a vue au-dessus, avec son entrée, sa sortie, son unité de traitement, sa mémoire, etc.

Organisation

Notre ordinateur contient donc :

- des **entrées** et des **sorties**, pour communiquer avec l'extérieur ;
- un truc qui va effectuer les instructions du programme : le **processeur** ;
- un machin qui va conserver nos données et le programme : la **mémoire** ;
- et enfin, de quoi faire communiquer le tout : le **Bus**.



Cela ressemble fortement à l'organisation vue plus haut, avec son entrée, sa sortie, son unité de traitement et sa mémoire. Rien d'étonnant à cela, notre ordinateur est un automate comme un autre, et il n'est pas étonnant qu'il reprenne une organisation commune à pas mal d'automates (mais pas à tous : certains fusionnent la mémoire et l'unité de traitement dans un seul gros circuit, ce que ne font pas nos ordinateurs). Rien n'empêche à notre ordinateur (ou à tout autre automate d'ailleurs) d'utiliser plusieurs processeurs, plusieurs mémoires, plusieurs bus, plusieurs entrées ou plusieurs sorties.

Périphériques

Cet ensemble de composants, ainsi que la façon dont ils communiquent entre eux est la structure minimum que tout ordinateur possède, le minimum syndical. Tout ce qui n'appartient pas à la liste du dessus est obligatoirement connecté sur les ports

d'entrée-sortie et est appelé **périphérique**. On peut donner comme exemple le clavier, la souris, l'écran, la carte son, etc.

Microcontroleurs

Parfois, on décide de regrouper la mémoire, les bus, le CPU et les ports d'entrée-sortie dans un seul boîtier, histoire de rassembler tout cela dans un seul composant électronique nommé **microcontrôleur**. Dans certains cas, qui sont plus la règle que l'exception, certains périphériques sont carrément inclus dans le microcontrôleur ! On peut ainsi trouver dans ces microcontrôleurs, des compteurs, des générateurs de signaux, des convertisseurs numériques-analogiques... On trouve des microcontrôleurs dans les disques durs, les baladeurs mp3, dans les automobiles, et tous les systèmes embarqués en général. Nombreux sont les périphériques ou les composants internes à un ordinateur qui contiennent des microcontrôleurs.

Maintenant qu'on connait un peu mieux l'organisation de base, voyons plus en détail ces différents composants.

Mémoire

La mémoire est, je me répète, le composant qui se chargera de stocker notre programme à éxecuter, ainsi que les données qu'il va manipuler. Son rôle est donc de retenir que des données ou des instructions stockées sous la forme de suites de bits, afin qu'on puisse les récupérer et les traiter.

ROM et RWM

Pour simplifier grandement, on peut grossièrement classer nos mémoire en deux types : les ***Read Only Memory***, et les ***Read Write Memory***.

Pour les mémoires ROM (les *Read Only Memory*), on ne peut pas modifier leur contenu. On peut récupérer une donnée ou une instruction dans notre mémoire : on dit qu'on y accède en **lecture**. Mais on ne peut pas modifier les données qu'elles contiennent. On utilise de telles mémoires pour stocker des programmes ou pour stocker des données qui ne peuvent pas varier. Par exemple, votre ordinateur contient une mémoire ROM spéciale qu'on appelle le ***BIOS***, qui permet de démarrer votre ordinateur, le configurer à l'allumage, et démarrer votre système d'exploitation.

Quand aux mémoire RWM (les *Read Write Memory*), on peut accéder à celle-ci en lecture, et donc récupérer une donnée stockée en mémoire, mais on peut aussi y accéder en **écriture** : on stocke une donnée dans la mémoire, ou on modifie une donnée existante. Ces mémoires RWM sont déjà plus intéressantes, et on peut les utiliser pour stocker des données. On va donc forcément trouver au moins une mémoire RWM dans notre ordinateur.

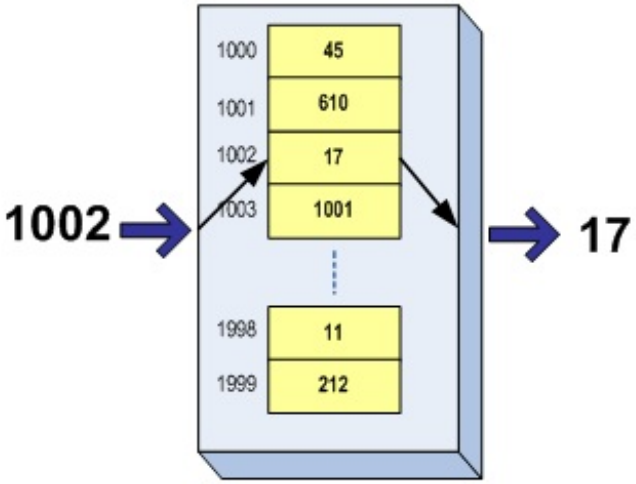
Pour l'anecdote, il n'existe pas de *Write Only Memory*.

Adressage

Pour utiliser cette mémoire, le processeur va pouvoir rapatrier des données depuis celle-ci. Pour éviter de s'emmêler les pinceaux, et confondre une donnée avec une autre, le processeur va devoir utiliser un moyen pour retrouver une donnée dans notre mémoire. Il existe plusieurs solutions, mais une de ces solutions est utilisée dans la grosse majorité des cas.

Dans la majorité des cas, notre mémoire est découpée en plusieurs **cases mémoires**, des blocs de mémoire qui contiennent chacun un nombre fini et constant de bits. Chaque case mémoire se voit attribuer un nombre binaire unique, l'**adresse**, qui va permettre de la sélectionner et de l'identifier celle-ci parmi toutes les autres. En fait, on peut comparer une adresse à un numéro de téléphone (ou à une adresse d'appartement) : chacun de vos correspondants a un numéro de téléphone et vous savez que pour appeler telle personne, vous devez composer tel numéro. Ben les adresses mémoires, c'est pareil !

Exemple : on demande à notre mémoire de sélectionner la case mémoire d'adresse 1002 et on récupère son contenu (ici, 17).



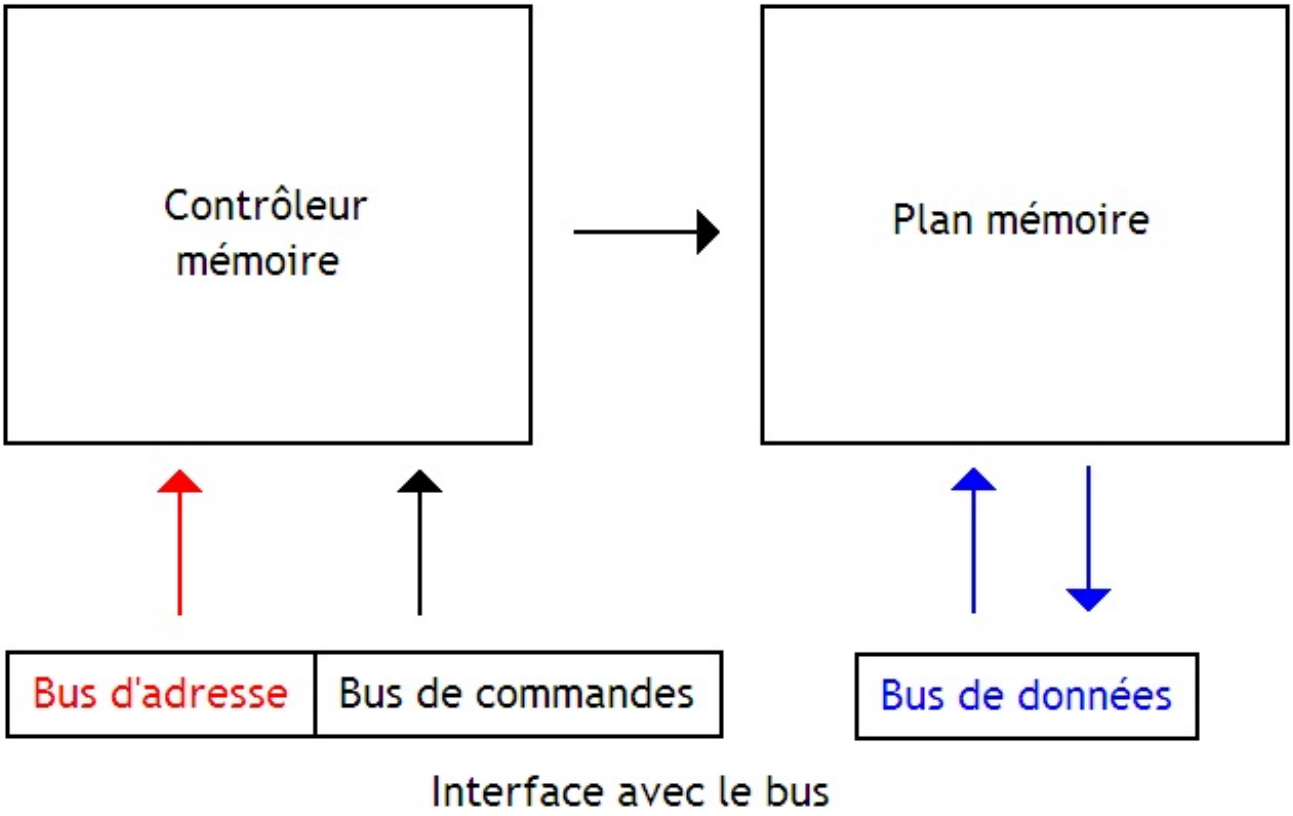
Il existe des mémoires qui ne fonctionnent pas sur ce principe, mais passons : ce sera pour la suite du tutoriel.

Anatomie

Une mémoire est un composant assez simple. Dans les grandes lignes, une mémoire est composée de deux à trois grands circuits. Le premier circuit contient toutes les cases mémoires : il s'agit du **plan mémoire**. C'est la mémoire proprement dite, là où sont stockées les données/instructions. Il existe différentes façons pour concevoir des cases mémoires. Pour information, dans le chapitre précédent, on avait vu comment créer des registres à partir de bascules D : on avait alors crée une case mémoire d'une mémoire RWM.

Ces cases mémoires ne nous servent à rien si l'on ne peut pas les sélectionner. Heureusement, les mémoires actuelles sont adressables, et on peut préciser quelle case mémoire lire ou écrire en précisant son adresse. Cette sélection d'une case à partie de son adresse ne se fait pas toute seule : on a besoin de circuits supplémentaires pour gérer l'adressage. Ce rôle est assuré par un circuit spécialisé qu'on appelle le **contrôleur mémoire**.

Et enfin, on doit relier notre mémoire au reste de l'ordinateur via un bus. On a donc besoin de **connexions avec le bus**. Ces connexions nous permettent aussi de savoir dans quel sens transférer les données (pour une mémoire RWM).



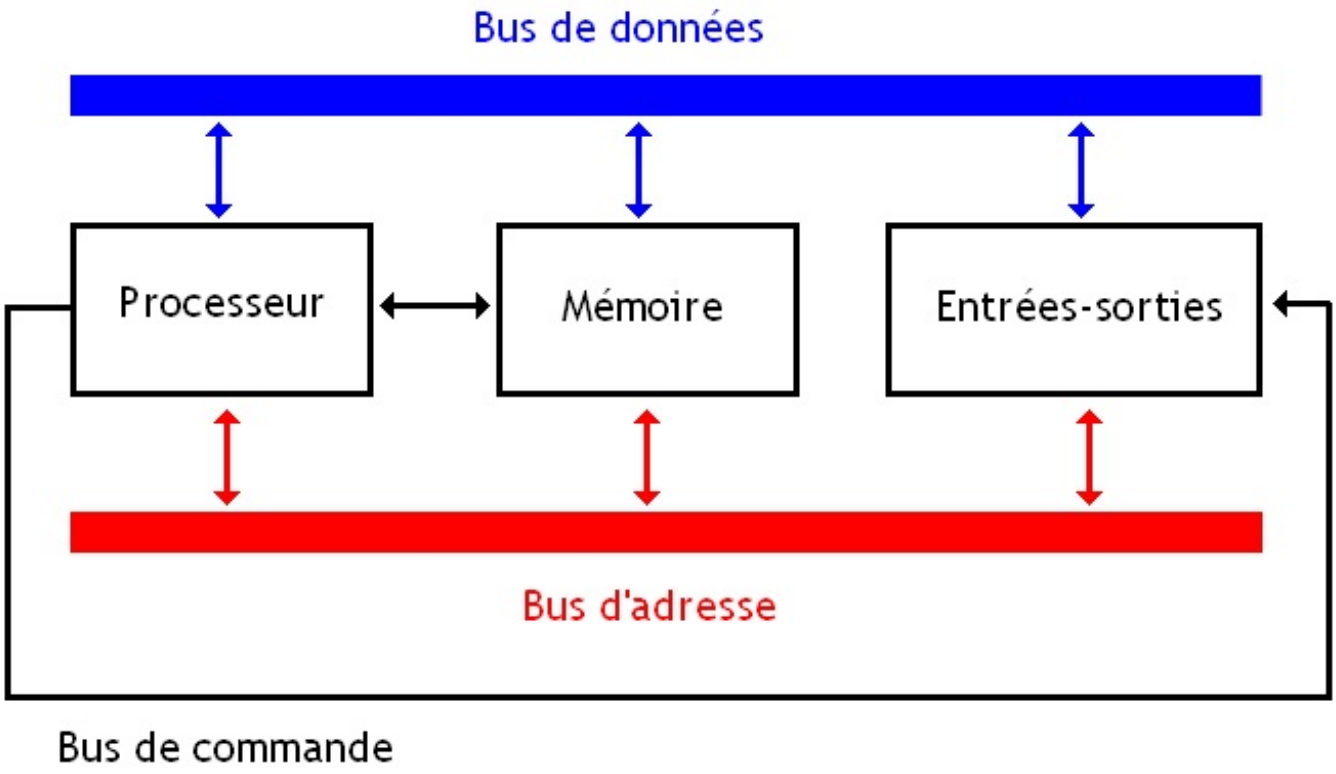
Bus de communication

Maintenant qu'on a une mémoire ainsi que nos entrées-sorties, il va bien falloir que notre processeur puisse les utiliser. Pour cela, le processeur est relié à la mémoire ainsi qu'aux entrées-sorties par un ou plusieurs **bus**. Ce bus n'est rien d'autre qu'un ensemble de fils électriques sur lesquels on envoie des zéros ou des uns. Ce bus relie le processeur, la mémoire, les entrées et les sorties ; et leur permet d'échanger des données ou des instructions.

Pour permettre au processeur (ou aux périphériques) de communiquer avec la mémoire, il y a trois prérequis que ce bus doit respecter :

- pouvoir sélectionner la case mémoire (ou l'entrée-sortie) dont on a besoin,
- préciser à la mémoire s'il s'agit d'une lecture ou d'une écriture,
- et enfin pouvoir transférer la donnée.

Pour cela, on doit donc avoir trois bus spécialisés, bien distincts, qu'on nommera le bus de commande, le bus d'adresse, et le bus de donnée. Ceux-ci relieront les différents composants comme indiqué dans le schéma qui suit.



Vous l'avez surement déjà deviné grâce à leur nom, mais je vais quand même expliquer à quoi servent ces différents bus.

Bus	Utilité
Bus d'adresse	Le **bus d'adresse** permet au processeur de sélectionner l'entrée, la sortie ou la portion de mémoire avec qui il veut échanger des données.
Bus de donnée	Le **bus de donnée** est un ensemble de fils par lequel s'échangent les données (et parfois les instructions) entre le processeur et le reste de la machine.
Bus de commande	Ce **bus de commande** va permettre de gérer l'intégralité des transferts entre la mémoire et le reste de l'ordinateur. Il peut transférer au moins un bit précisant si on veut lire ou écrire dans la mémoire. Généralement, on considère par convention que ce bit vaut : • 1 si on veut faire une lecture, • 0 si c'est pour une écriture.

Processeur

C'est un composant qui va prendre en entrée une ou plusieurs données et éxecuter des instructions. Ces instructions peuvent être des additions, des multiplications, par exemple, mais qui peuvent aussi faire des choses un peu plus utiles. Ce processeur est aussi appelé ***Central Processing Unit***, abrévié en **CPU**.

Un processeur ne peut qu'effectuer une suite d'instructions dans un ordre bien précis. C'est cette propriété qui fait que notre ordinateur est un automate particulier, programmable : on lui permet de faire des instructions indépendantes, et on peut organiser ces instructions dans l'ordre que l'on souhaite : en clair, créer un programme. Pour vous donner une idée de ce que peut être une instruction, on va en citer quelques-unes.

Instructions arithmétiques

Les instructions les plus communes sont des **instructions arithmétiques et logiques**, qui font simplement des calculs sur des nombres. On peut citer par exemple :

- ***ET*** logique entre deux nombres (consiste à effectuer un ***ET*** entre les bits de même rang de deux nombres) ;
- ***OU*** logique entre deux nombres (consiste à effectuer un ***OU*** entre les bits de même rang de deux nombres) ;
- ***NON*** logique : inverse tous les bits d'un nombre ;
- ***XOR*** logique entre deux nombres (consiste à effectuer un ***XOR*** entre les bits de même rang de deux nombres) ;
- addition de deux nombres;
- multiplication ;
- division ;
- modulo ;
- soustraction ;
- ...

Ces instructions sont des instructions dont le résultat ne dépend que des données à traiter. Elle sont généralement prises en charge par un circuit combinatoire indépendant, qui s'occupe exclusivement du calcul de ces instructions : **l'unité de calcul**.

Registres

Pour pouvoir fonctionner, tout processeur va devoir stocker un certain nombre d'informations nécessaires à son fonctionnement : il faut qu'il se souvienne à quel instruction du programme il en est, qu'il connaisse la position en mémoire des données à manipuler, qu'il manipule certaines données, etc. Pour cela, il contient des **registres**. Ces registres sont de petites mémoires ultra-rapides fabriquées avec des bascules.

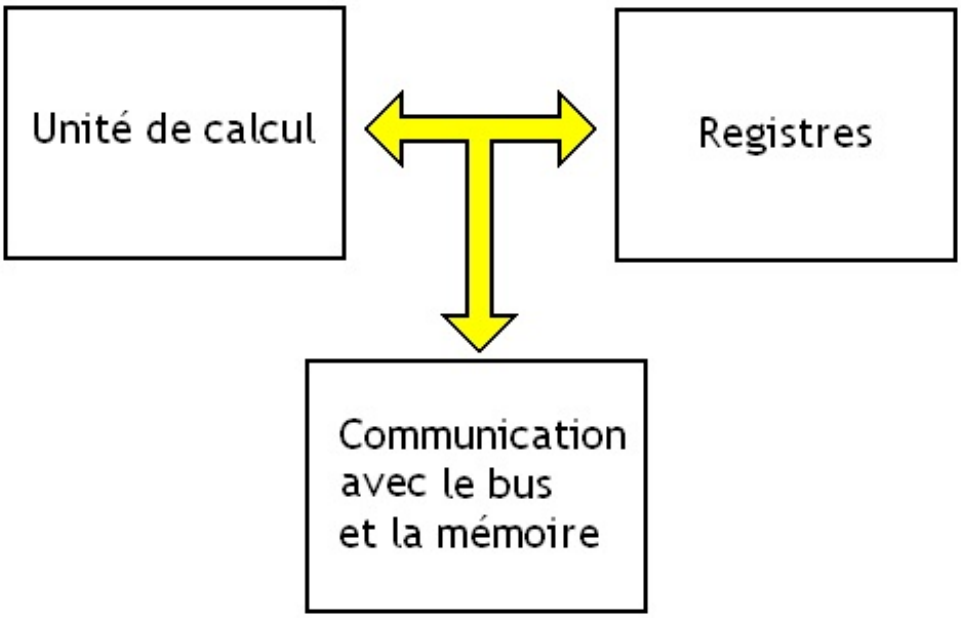
Ces registres peuvent servir à plein de choses : stocker des données afin de les manipuler plus facilement, stocker l'adresse de la prochaine instruction, stocker l'adresse d'une donnée à aller chercher en mémoire, etc. Bref, suivant le processeur, ces registres peuvent servir à tout et n'importe quoi.

Instructions d'accès mémoire

Pour faire ces calculs, et exécuter nos instructions arithmétiques et logiques, notre processeur doit aller chercher les données à manipuler dans la mémoire RAM ou dans ses registres. Après tout, les données manipulées par nos instructions ne sortent pas de nulle part. Certaines d'entre elles peuvent être stockées dans les registres du processeur, mais d'autres sont stockées dans la mémoire principale : il faut bien y aller les chercher.

Pour cela, notre processeur va devoir échanger des données entre les registres et la mémoire, copier une donnée d'un endroit de la mémoire à un autre, copier le contenu d'un registre dans un autre, modifier directement le contenu de la mémoire, effectuer des lectures ou écriture en mémoire principale, etc.

Comme vous l'avez surement deviné, les accès mémoires ne sont pas pris en charge par les unités de calcul. Pour gérer ces communications avec la mémoire, le processeur devra être relié à la mémoire et devra décider quoi lui envoyer comme ordre sur les différents bus (bus de commande, de donnée, d'adresse). Il pourra ainsi lui envoyer des ordres du style : "Je veux récupérer le contenu de l'adresse X", ou "Enregistre moi la donnée que je t'envoie à l'adresse Y". Ces ordres seront transmis via le bus. L'intérieur de notre processeur ressemble donc à ceci, pour le moment :



Program Counter

Il est évident que pour exécuter une suite d'instructions dans le bon ordre, notre ordinateur doit savoir quelle est la prochaine instruction à exécuter. Il faut donc que notre processeur se souvienne de cette information quelque part : notre processeur doit donc contenir une mémoire qui stocke cette information. C'est le rôle du registre d'adresse d'instruction, aussi appelé ***Program Counter***.

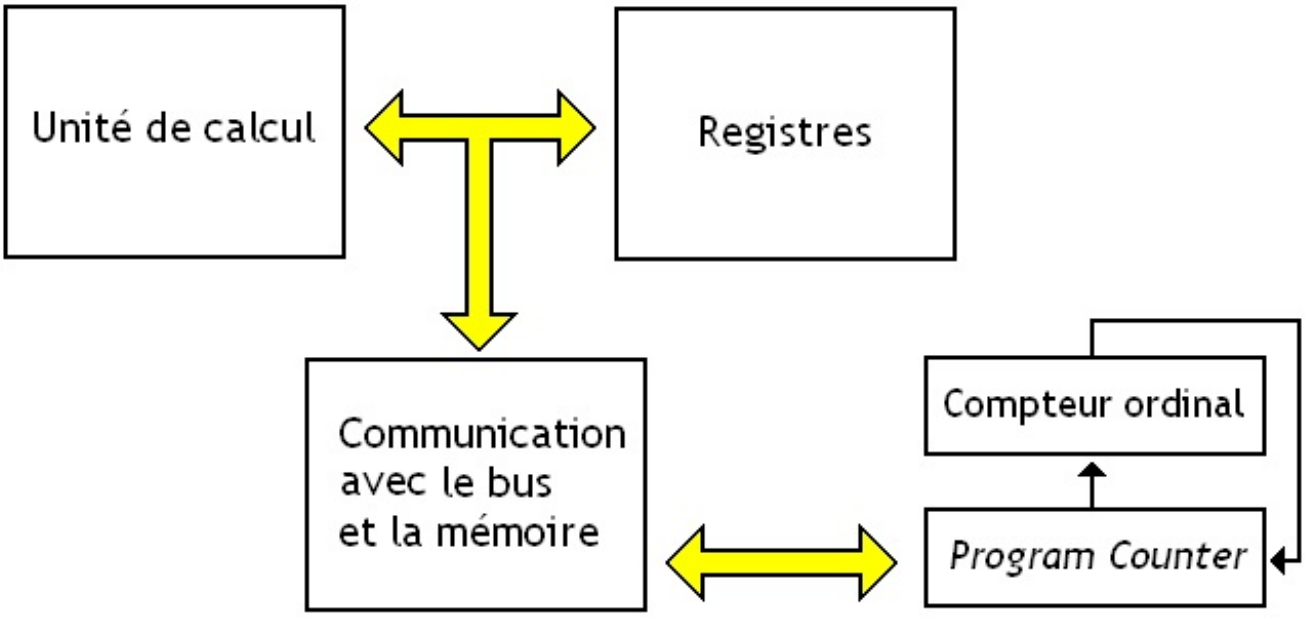
Ce registre stocke l'adresse de la prochaine instruction à exécuter. Cette adresse permet de localiser l'instruction suivante en mémoire. Cette adresse ne sort pas de nulle part : on peut la déduire de l'adresse de l'instruction en cours d'exécution par divers moyens plus ou moins simples qu'on verra dans la suite de ce tutoriel.

Ce calcul peut être fait assez simplement. Généralement, on profite du fait que ces instructions sont exécutées dans un ordre bien précis, les unes après les autres. Sur la grosse majorité des ordinateur, celles-ci sont placées les unes à la suite des autres dans l'ordre où elles doivent être exécutées. L'ordre en question est décidé par le programmeur. Un programme informatique n'est donc qu'une vulgaire suite d'instructions stockée quelque part dans la mémoire de notre ordinateur.

Adresse	Instruction
0	Charger le contenu de l'adresse 0F05
1	Charger le contenu de l'adresse 0555
2	Additionner ces deux nombres
3	Charger le contenu de l'adresse 0555
4	Faire en XOR avec le résultat antérieur
...	...
5464	Instruction d'arrêt

En faisant ainsi, on peut calculer facilement l'adresse de la prochaine instruction en ajoutant la longueur de l'instruction juste chargée (le nombre de case mémoire qu'elle occupe) au contenu du registre d'adresse d'instruction. Dans ce cas, l'adresse de la prochaine instruction est calculée par un petit circuit combinatoire couplé à notre registre d'adresse d'instruction, qu'on appelle le **compteur ordinal**.

L'intérieur de notre processeur ressemble donc plus à ce qui est indiqué dans le schéma du dessous.



Mais certains processeurs n'utilisent pas cette méthode. Sur de tels processeurs, chaque instruction va devoir préciser quelle est

la prochaine instruction. Pour ce faire, une partie de la suite de bit représentant notre instruction à exécuter va stocker cette adresse. Dans ce cas, ces processeurs utilisent toujours un registre pour stocker cette adresse, mais ne possèdent pas de compteur ordinal, et n'ont pas besoin de calculer une adresse qui leur est fournie sur un plateau.

Prise de décision

Notre processeur peut donc exécuter des instructions les unes à la suite des autres grâce à notre registre d'adresse d'instruction (le *Program* Counter). C'est bien, mais on ne pas bien loin avec ce genre de choses. Il serait évidemment mieux si notre processeur pouvait faire des choses plus évoluées et s'il pouvait plus ou moins s'adapter aux circonstances au lieu de réagir machinalement. Par exemple, on peut souhaiter que celui-ci n'exécute une suite d'instructions que si une certaine condition est remplie et ne l'exécute pas sinon. Ou faire mieux : on peut demander à notre ordinateur de répéter une suite d'instructions tant qu'une condition bien définie est respectée.

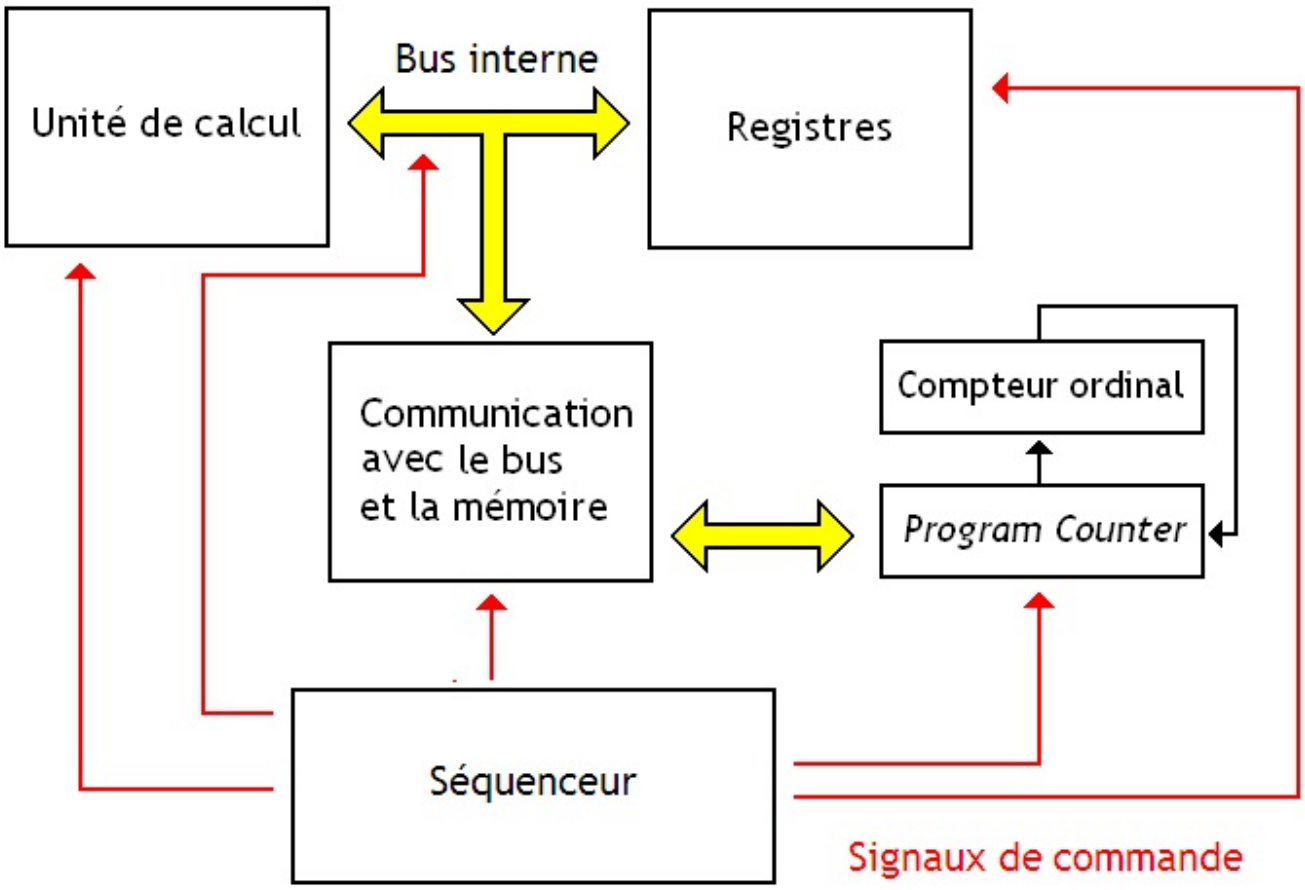
Pour ce faire, on a crée des instructions un peu spéciales, qui permettent de "sauter" directement à une instruction dans notre programme, et poursuivre l'exécution à partir de cette instruction. Cela permet au programme de passer directement à une instruction située plus loin dans le déroulement normal du programme, voir de revenir à une instruction antérieure. Ces instructions sont ce qu'on appelle des **branchements**.

Pour ce faire, elles modifient le contenu du registre d'adresse d'instruction, et y place l'adresse de l'instruction à laquelle on veut sauter. Ces instructions sont appelées des branchements. Elles sont très utiles pour créer nos programmes informatiques, et il serait vraiment difficle, voire impossible de vous passer d'elles. Tout programmeur utilise des branchements quand il programme : il ne s'en rend pas compte, mais ces branchements sont souvent cachés derrière des fonctionnalités basiques de nos langages de programmation usuels (les if, tests, boucles, et fonctions sont fabriquées avec des branchements).

Séquenceur

Quoiqu'il en soit, toutes nos instructions sont stockées en mémoire sous la forme de suites de bits. A telle instruction correspondra telle suite de bit. Notre processeur devra donc décider quoi faire de ces suites de bits, et les interpréter, en déduire quoi faire. Par exemple, est-ce que la suite de bit que je viens de lire me demande de charger une donnée depuis la mémoire, est-ce qu'elle me demande de faire une instruction arithmétique, etc. Une fois cela fait, il faut ensuite aller configurer la mémoire pour gérer les instructions d'accès mémoire (lire la bonne adresse, préciser le sens de transferts, etc), ou commander l'unité de calcul afin qu'elle fasse une addition et pas une multiplication, ou mettre à jour le registre d'adresse d'instruction si c'est un branchement, etc.

Pour ce faire, notre processeur va contenir un circuit séquentiel spécial, qui déduit quoi faire de la suite d'instruction chargée, et qui commandera les circuits du processeur. Ce circuit spécialisé s'appelle le **séquenceur**.


Unité de calcul
Bus interne
Registres
Communication
avec le bus
et la mémoire
Compteur ordinal
Program Counter
Séquenceur
Signaux de commande

La gestion de la mémoire

On a vu que notre programme était stocké dans la mémoire de notre ordinateur. Les instructions du programme exécuté par le processeur sont donc stockées comme toutes les autres données : sous la forme de suites de bits dans notre mémoire, tout comme les données qu'il va manipuler. Dans ces conditions, difficile de faire la différence entre donnée et instruction. Mais rassurez-vous : le processeur intègre souvent des fonctionnalités qui empêchent de confondre une donnée avec une instruction quand il va chercher une information en mémoire.

Ces fonctionnalités ne sont pas totalement fiables, et il arrive assez rarement que le processeur puisse confondre une instruction ou une donnée, mais cela est rare. Cela peut même être un effet recherché : par exemple, on peut créer des programmes qui modifient leurs propres instructions : cela s'appelle du ***self modifying code***, ce qui se traduit par code automodifiant en français. Ce genre de choses servait autrefois à écrire certains programmes sur des ordinateurs rudimentaires (pour gérer des tableaux et autres fonctionnalités de base utilisées par les programmeurs), pouvait aussi permettre de rendre nos programmes plus rapides, servait à compresser un programme, ou pire : permettait de cacher un programme et le rendre indétectable dans la mémoire (les virus informatiques utilisent beaucoup de genre de procédés). Mais passons !

Deux mémoires pour le prix d'une

La plus importante de ces astuces évitant la confusion entre données et instructions est très simple : les instructions et les données sont stockées dans deux portions de mémoire bien séparées. Sur de nombreux ordinateurs, la mémoire est séparée en deux gros blocs de mémoires bien spécialisés :

- un bloc de mémoire qui stocke le programme, nommée la **mémoire programme** ;
- et un bloc de mémoire qui stocke le reste nommée la **mémoire travail**.

Ces blocs de mémoire vont donc stocker des contenus différents :

Portion de la mémoire	Mémoire programme	Mémoire de travail
Contenu du bloc	• le **programme** informatique à exécuter • et parfois les **constantes** : ce sont des données qui peuvent être lues mais ne sont jamais accédées en écriture durant l'exécution du programme. Elle ne sont donc jamais modifiées et gardent la même valeur quoi qu'il se passe lors de l'exécution du programme.	les **variables** du programme à exécuter, qui sont des données que le programme va manipuler.

Il faut toutefois préciser que ce découpage en mémoire programme et mémoire de travail n'est pas une obligation. En effet, certains ordinateurs s'en passent complétement : je pense notamment aux architectures *dataflow*, une classe d'ordinateur assez spéciale, qui ne sera pas traitée dans ce tutoriel, mais qui est néanmoins abordées dans un article assez compliqué sur ce site. Mais remettons cela à plus tard, pour quand vous aurez un meilleur niveau.

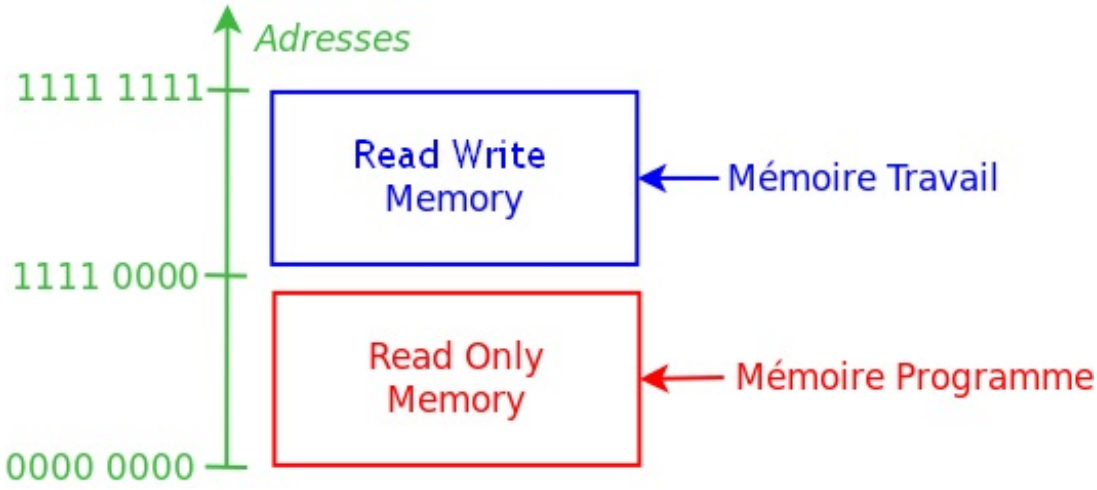
Le processeur ne traitera pas de la même façon les instructions en mémoire programme et les données présentes en mémoire de travail, afin de ne pas faire de confusions. Nos instructions sont en effet interprétées par le séquenceur, tandis que nos données sont manipulées par l'unité de calcul. Et tout cela, c'est grâce à l'existence du *Program Counter*, le fameux registre d'adresse d'instruction vu précédemment. En regroupant nos instructions dans un seul bloc de mémoire, et en plaçant nos instructions les unes à la suite des autres, on est sur que le registre d'adresse d'instruction passera d'une instruction à l'autre en restant dans un bloc de mémoire ne contenant que des instructions. Sauf s'il déborde de ce bloc, ou qu'un branchement renvoie notre processeur n'importe où dans la mémoire, mais passons.

Quoiqu'il en soit, ce découpage entre mémoire programme et mémoire de travail est quelque chose d'assez abstrait qui peut être mis en pratique de différentes manières. Sur certains ordinateurs, on utilise deux mémoires séparées : une pour le programme, et une pour les données. Sur d'autres, on utilisera une seule mémoire, dont une portion stockera notre programme, et l'autre servira de mémoire de travail. Il faut bien faire la différence entre le découpage de notre mémoire en mémoires de programmes et de travail, purement "conceptuelles" ; et les différentes mémoires qu'on trouvera dans nos ordinateurs. Rassurez-vous, vous allez comprendre en lisant la suite. Dans ce qui suit, on va voir comment des deux mémoires sont organisées dans nos ordinateurs.

Séparation matérielle des mémoires

Sur les ordinateurs très simples, La mémoire programme et la mémoire travail sont souvent placées dans deux mémoires séparées. Il y a deux composants électroniques, chacun dans un boîtier séparé : un pour la mémoire programme et un autre pour la mémoire travail.

Avec cette séparation dans deux mémoires séparées, la mémoire programme est généralement une mémoire de type **ROM**, c'est à dire accessible uniquement en lecture : on peut récupérer les informations conservées dans la mémoire (on dit qu'on effectue une lecture), mais on ne peut pas les modifier. Par contre, la mémoire travail est une mémoire RWM : on peut lire les informations conservées, mais on peut aussi modifier les données qu'elle contient (écriture). On peut ainsi effectuer de nombreuses manipulations sur le contenu de cette mémoire : supprimer des données, en rajouter, les remplacer, les modifier, etc.

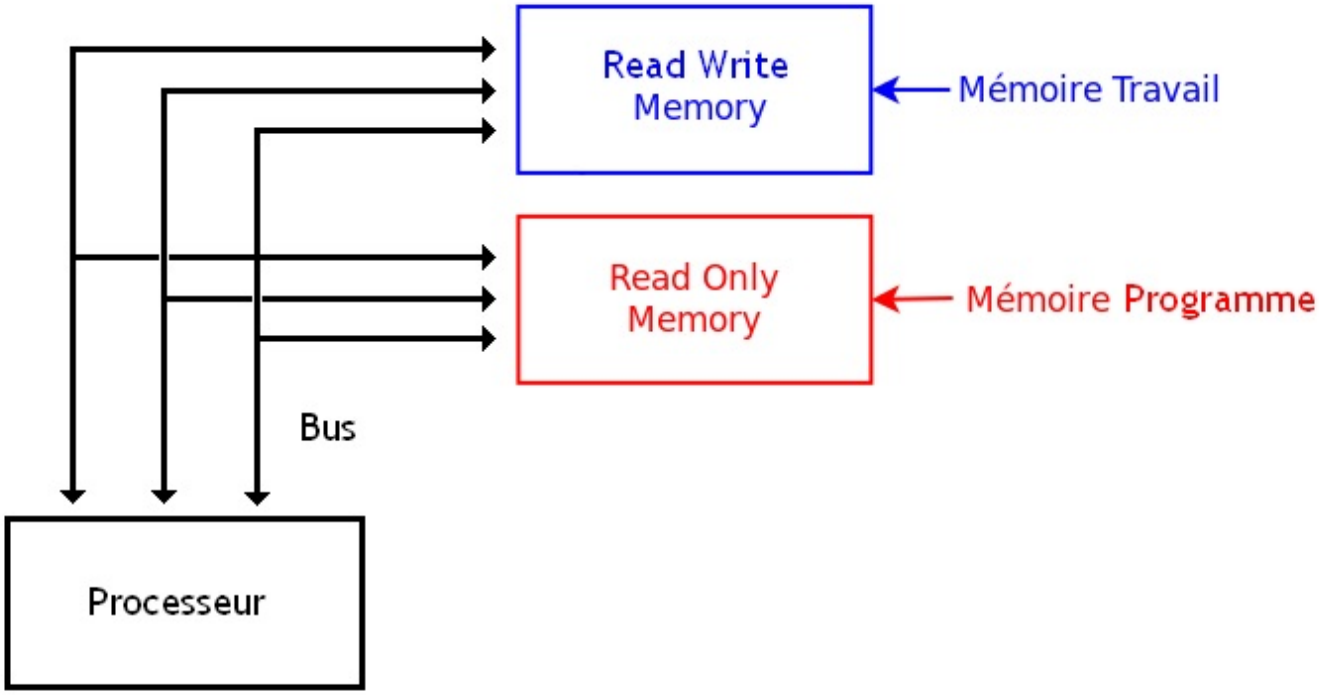


Architectures Harvard et Von Neumann

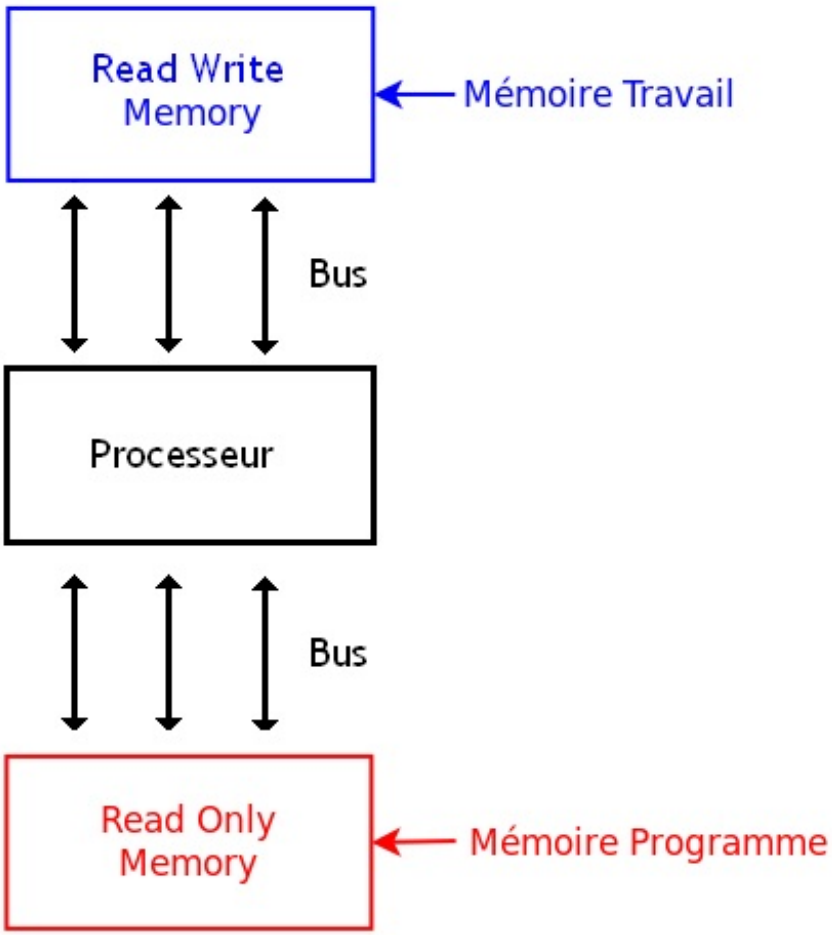
On a vu que le processeur est relié à la mémoire par un ensemble de fils qui connectent ces deux composants, le bus. Dans le cas où la mémoire programme et la mémoire travail sont séparées dans deux composants électroniques matériellement différents, il y a deux façon de relier ces deux mémoires au processeur par un bus :

- un seul bus pour les deux mémoires.
- un bus par mémoire.

Le premier cas s'appelle l'**architecture Von Neumann**.



Le second s'appelle l'**architecture Harvard**.

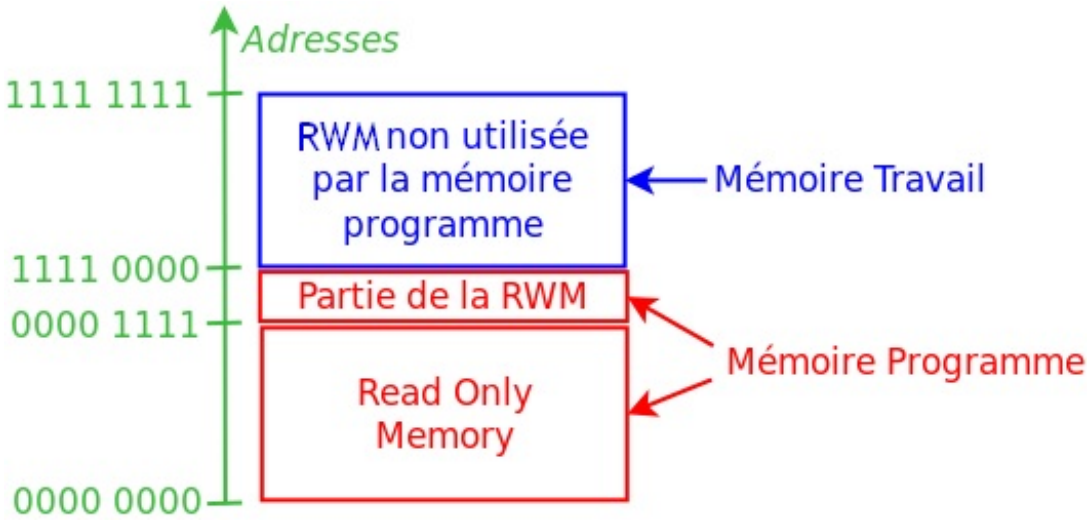


Chacune possède quelques avantages et inconvénients:

	Architecture Von neumann	Architecture Harvard
Avantages	• Accès à la mémoire facile : un seul bus à gérer ; • Un seul bus à câbler : simplicité de conception.	• Permet de charger une instruction et une donnée simultanément : on charge la donnée sur le bus qui relie la mémoire de travail au processeur, et l'instruction sur le bus qui relie processeur et mémoire programme. Les deux bus étant séparés, on peut le faire simultanément. On se retrouve donc avec un gain de vitesse
Inconvénients	• Ne peut pas charger une donnée simultanément avec une instruction : on doit charger la donnée, puis l'instruction, vu que tout passe par un seul bus. Ce genre d'architecture est donc plus lente.	• Deux bus à câbler et à gérer ; • Accès à la mémoire plus compliqué à gérer.

Architecture modifiée

Sur d'autres, on a besoin de modifier certains paramètres du programmes pour qu'il s'adapte à certaines circonstances. Pour ce faire, il faut donc modifier certaines parties de la mémoire programme. On ne peut donc stocker ces paramètres en ROM, et on préfère plutôt les stocker dans une RWM : la mémoire programme est donc composée d'une ROM et d'une partie de la RWM.



Avec cette organisation, une partie ou la totalité du programme est stocké dans une mémoire censée stocker des données. Rien de choquant à cela : programme et données sont tous les deux stockés sous la forme de suites de bits dans la mémoire. Rien n'empêche de copier l'intégralité du programme de la mémoire ROM vers la mémoire RWM, mais ce cas est assez rare.

Mettre les programmes sur un périphérique

On peut même aller plus loin : on peut utiliser une mémoire ROM, contenant un programme de base, et charger directement nos programmes dans la mémoire RWM, depuis un périphérique connecté sur une entrée-sortie : un disque dur, par exemple. Dans ce cas, la mémoire programme n'est pas intégralement stockée dans une ROM : le programme est en effet placé sur un périphérique et chargé en mémoire RWM pour être exécuté. Mais il y a toujours dans tous les ordinateurs, une petite mémoire ROM. Cette ROM contient un petit programme qui va charger le programme stocké sur le périphérique dans la mémoire de travail.
On aura donc le système d'exploitation et nos programmes qui seront donc copiés en mémoire RWM :

- une partie de la mémoire RWM deviendra la mémoire programme qui stockera vos applications et le système d'exploitation
- et une autre restera de la mémoire travail.

L'avantage, c'est qu'on peut modifier le contenu d'un périphérique assez facilement, tandis que ce n'est pas vraiment facile de modifier le contenu d'une ROM (et encore, quand c'est possible). On peut ainsi facilement installer ou supprimer des programmes sur notre périphérique, en rajouter, en modifier, les mettre à jour sans que cela ne pose problème. C'est cette solution qui est utilisée dans nos PC actuels, et la petite mémoire ROM en question s'appelle le **BIOS**.

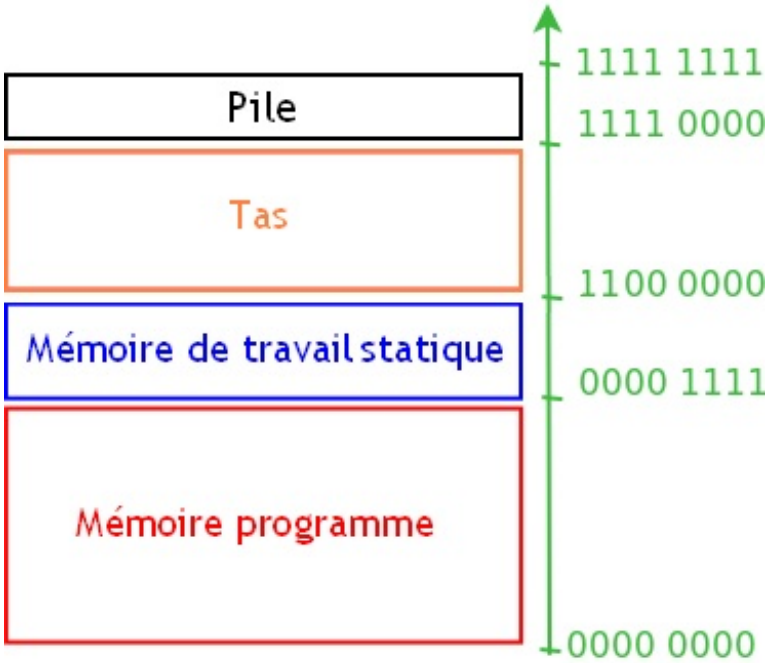
L'organisation de la mémoire et la pile

Reste que notre mémoire de travail peut-être organisée de différentes façons, et que celle-ci est elle-même subdivisée en plusieurs morceaux de taille et d'utilité différentes. Suivant le programme que vous utilisez, ou votre système d'exploitation, la mémoire est généralement organisée plus ou moins différemment : votre système d'exploitation ou le programme exécuté peut ainsi réserver certains morceau de programme pour telle ou telle fonctionnalité, ou pour stocker des données particulières. Mais certaines particularités reviennent souvent.

Pile, Tas et Mémoire Statique

Généralement, la mémoire d'un ordinateur est segmentée en quatre parties. On retrouve la **mémoire programme**, contenant le programme. Par contre, notre mémoire de travail est découpée en trois portions, qui ont des utilités différentes :

- la **mémoire de travail statique** ;
- le **tas** ;
- et la **pile**.



La mémoire de travail statique est une partie de la mémoire de travail dans laquelle on stocke des données définitivement. En clair, on ne peut pas supprimer l'espace mémoire utilisé par une donnée dont on n'a plus besoin pour l'utiliser pour stocker une autre donnée. On peut donc lire ou modifier la valeur d'une donnée, mais pas la supprimer. Et c'est pareil pour la mémoire programme : on ne peut pas supprimer tout un morceau de programme en cours d'exécution (sauf dans quelques cas particuliers vraiment tordus).

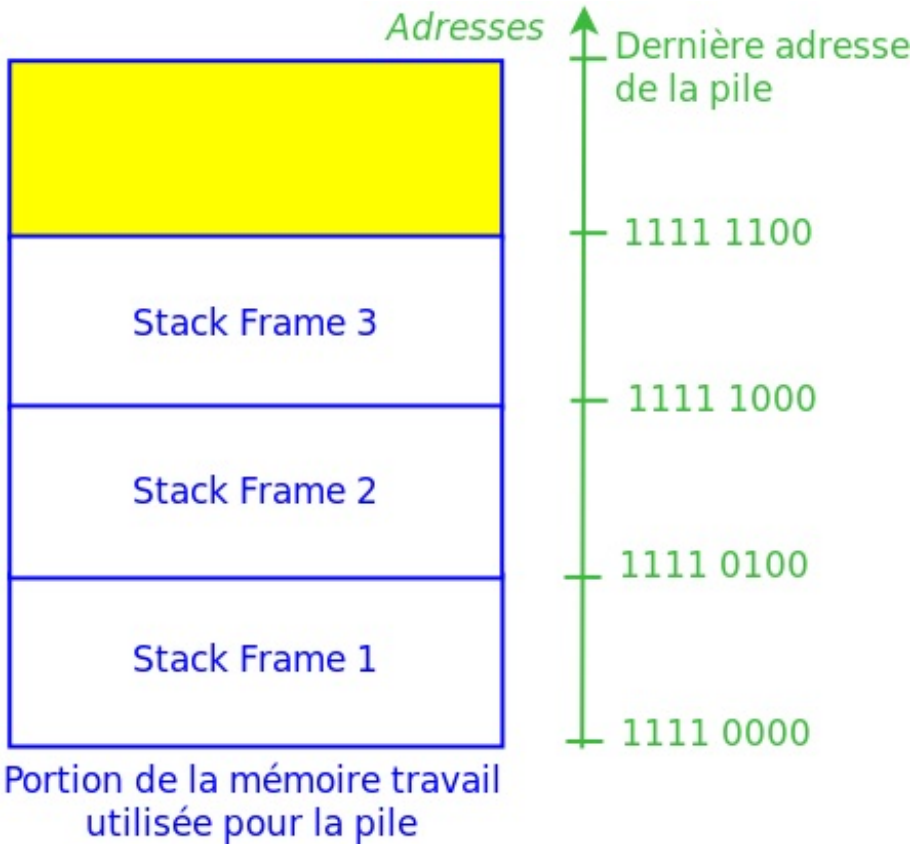
A l'inverse, on peut utiliser le reste de la mémoire pour stocker temporairement des données et les effacer lorsqu'elles deviennent inutiles. Cela permet de limiter l'utilisation de la mémoire. Cette partie de la mémoire utilisable au besoin peut être utilisée de deux façon :

- Soit avec une **pile**
- Soit avec un **tas**.

La différence principale entre le tas et la pile est la façon dont sont organisées les données dedans. Une autre différence est leur utilisation : le tas est intégralement géré par le logiciel (par le programme en cours d'exécution et éventuellement le système d'exploitation), tandis que la pile est en partie, voire totalement, gérée par le matériel de notre ordinateur. Dans ce qui va suivre, on va parler de la pile. Pourquoi ? Et bien parce que celle-ci est en partie gérée par notre matériel, et que certains processeurs l'utilisent abondamment. Il existe même des processeurs qui utilisent systématiquement cette pile pour stocker les données que notre processeur doit manipuler. Ces processeurs sont appelés des machines à pile, ou ***stack machines***.

La pile

Comme je l'ai dit plus haut, la pile est une partie de la mémoire de travail. Mais cette portion de la RAM a une particularité : on stocke les données à l'intérieur d'une certaine façon. Les données sont regroupées dans la pile dans ce qu'on appelle des ***stack frame*** ou cadres de pile. Ces *stack frames* regroupent plusieurs cases mémoires contiguës (placées les unes à la suite des autres). On peut voir ces *stack frames* comme des espèces de blocs de mémoire.



Sur les *stack machines*, ces *stack frames* stockent généralement un nombre entier, des adresses, des caractères, ou un nombre flottant ; mais ne contiennent guère plus. Mais sur d'autres processeurs un peu plus évolués, on utilise la pile pour stocker autre chose, et il est alors nécessaire d'avoir des *stack frame* pouvant stocker des données plus évoluées, voire stocker plusieurs données hétérogènes dans une seule *stack frame*. Ce genre de choses est nécessaire pour implémenter certaines fonctionnalités de certains langages de haut niveau.

Last Input First Output

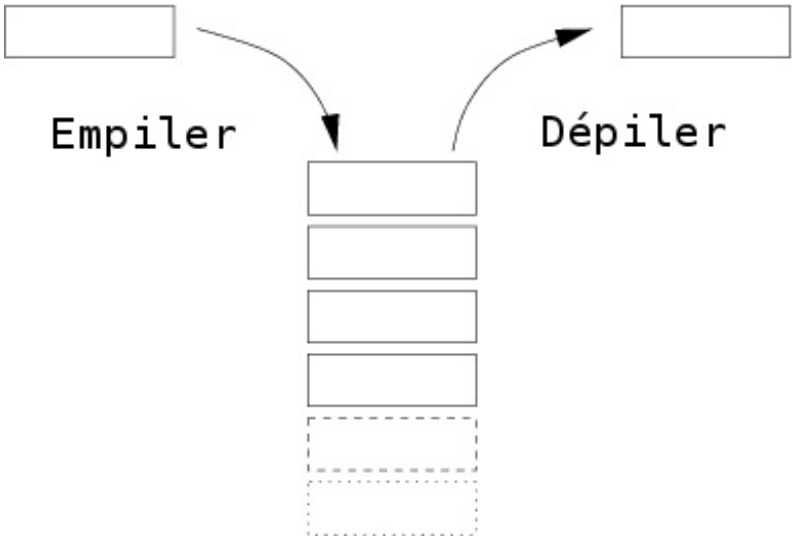
Mais ce qui différencie une pile d'une simple collection de morceaux de mémoire, c'est la façon dont les *stack frames* sont gérées.

Comme on peut le voir facilement, les *stack frame* sont crées une par unes, ce qui fait qu'elles sont placées les unes à la suite des autres dans la mémoire : on crée une *stack frame* immédiatement après la précédente. C'est une première contrainte : on ne peut pas créer de *stack frames* n'importe où dans la mémoire. On peut comparer l'organisation des *stack frames* dans la pile à une pile d'assiette : on peut parfaitement rajouter une assiette au sommet de la pile d'assiette, ou enlever celle qui est au sommet, mais on ne peut pas toucher aux autres assiettes. Sur la pile de notre ordinateur, c'est la même chose : on ne peut accéder qu'à la donnée située au sommet de la pile. Comme pour une pile d'assiette, on peut rajouter ou enlever une *stack frame* au sommet de la pile, mais pas toucher aux *stack frame* en dessous, ni les manipuler.

Le nombre de manipulations possibles sur cette pile se résume donc à trois manipulations de base qu'on peut combiner pour créer des manipulations plus complexes.

On peut ainsi :

- détruire la *stack frame* au sommet de la pile, et supprimer tout son contenu de la mémoire : on **dépile**.
- créer une *stack frame* immédiatement après la dernière *stack frame* existante : on **empile**.
- utiliser les données stockées dans la *stack frame* au sommet de la pile.



Source de l'image : Wikipédia

Si vous regardez bien, vous remarquerez que la donnée au sommet de la pile est la dernière donnée à avoir été ajoutée (empilée) sur la pile. Ce sera aussi la prochaine donnée à être dépilée (si on n'empile pas de données au dessus). Ainsi, on sait que dans cette pile, les données sont dépilées dans l'ordre inverse d'empilement. Ainsi, la donnée au sommet de la pile est celle qui a été ajoutée le plus récemment.

Au fait, la pile peut contenir un nombre maximal de *stack frames*, ce qui peut poser certains problèmes. Si l'on souhaite utiliser plus de *stack frames* que possible, il se produit un *stack overflow*, appelé en français **débordement de pile**. En clair, l'ordinateur plante !

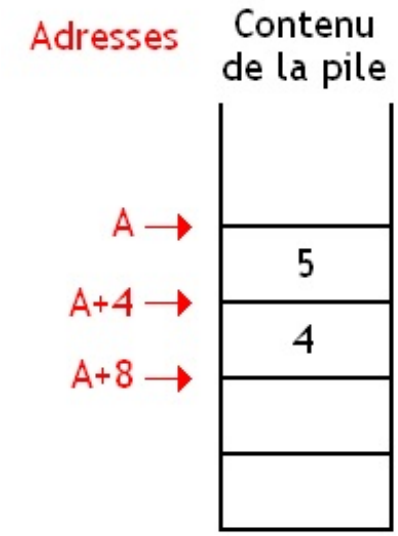
Machines à pile et successeurs

De ce qu'on vient de voir, on peut grosso-modo classer nos ordinateurs en deux grandes catégories : les **machines à pile**, et les **machines à accès aléatoire**.

Machines à pile

Les machines à pile, aussi appelées *stack machines* en anglais, utilisent la pile pour stocker les données manipulées par leurs instructions. Sur ces machines, les cadres de pile ne peuvent contenir que des données simples. Par données simples, il faut comprendre données manipulables de base par le processeur, comme des nombres, ou des caractères. Leur taille est donc facile à déterminer : elle est de la taille de la donnée à manipuler.

Exemple avec des entiers de 4 octets.



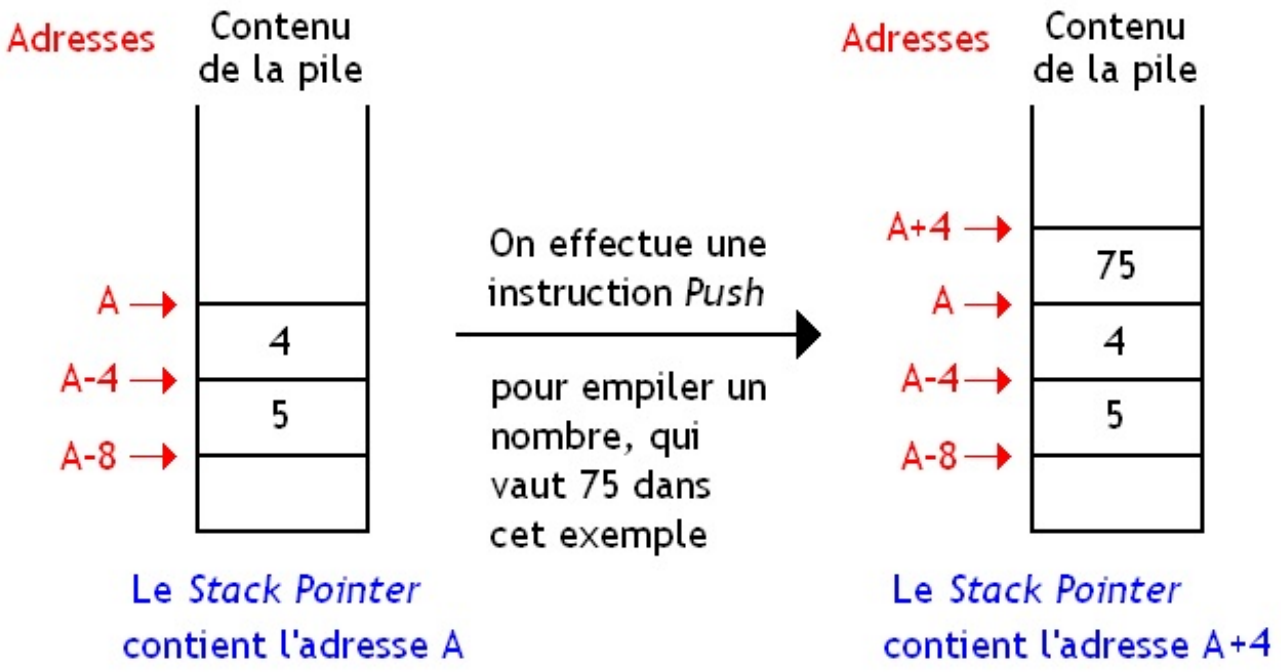
Ces machines ont besoin d'un registre pour fonctionner : il faut bien stocker l'adresse du sommet de la pile. Je vous présente donc le ***Stack Pointer***, qui n'est autre que ce fameux registre qui stocke l'adresse du sommet de la pile. Ce registre seul suffit : nos cadres de pile ayant une taille bien précise, on peut se passer de registre pour stocker leur taille ou leur adresse de début/fin

en se débrouillant bien.

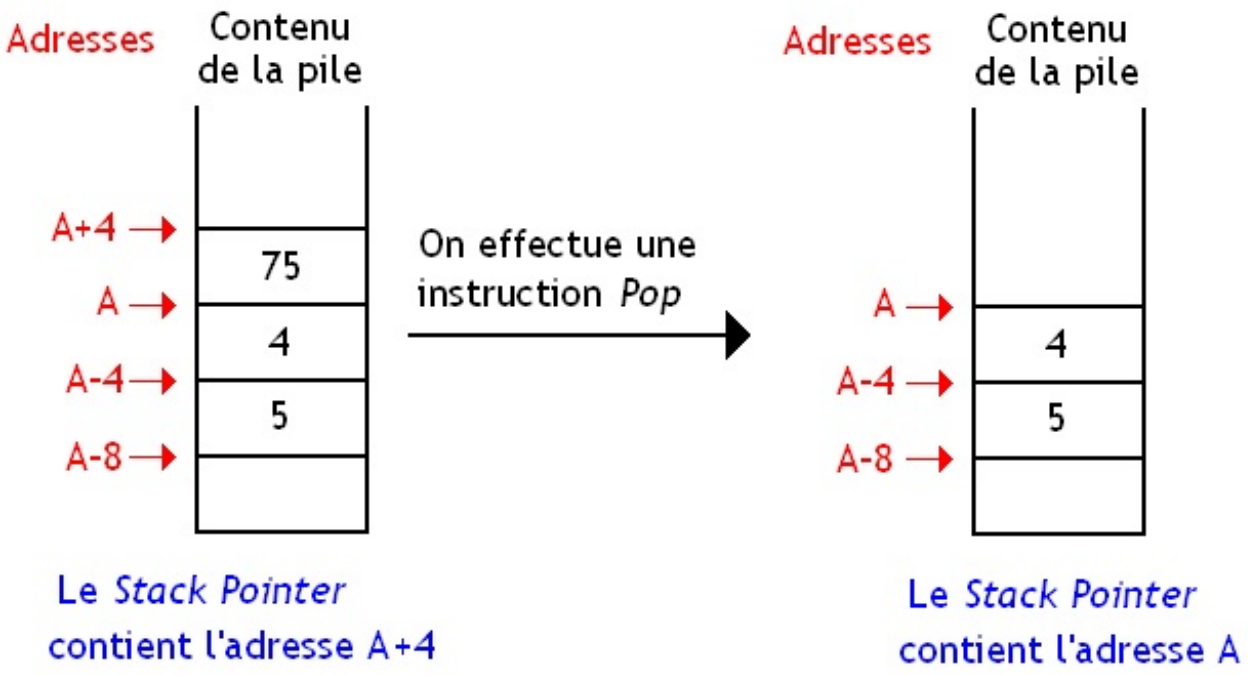
Sur certaines machines à pile très simples, la pile n'est pas tout à fait stockée dans une portion de la mémoire : elle est stockée directement dans le processeur. Le processeur contient ainsi un grand nombre de registres, qui seront utilisés comme une pile. Ces registres étant plus rapides que la mémoire principale de l'ordinateur, les opérations manipulant uniquement la pile et ne devant pas manipuler la mémoire seront donc beaucoup plus rapides (les autres instructions étant aussi accélérées, mais moins).

Push Et Pop

Bien évidemment, les données à traiter ne s'empilent pas toutes seules au sommet de la pile. Pour empiler une donnée au sommet de la pile, notre processeur fourni une instruction spécialement dédiée. Cette instruction s'appelle souvent `Push`. Elle permet de copier une donnée vers le sommet de la pile. Cette donnée peut être aussi bien dans la mémoire statique que dans le tas, peu importe. Cette instruction va prendre l'adresse de la donnée à empiler, et va la stocker sur la pile. Bien évidemment, le contenu du *Stack Pointer* doit être mis à jour : on doit additionner (ou soustraire, si on fait partir la pile de la fin de la mémoire) la taille de la donnée qu'on vient d'empiler.



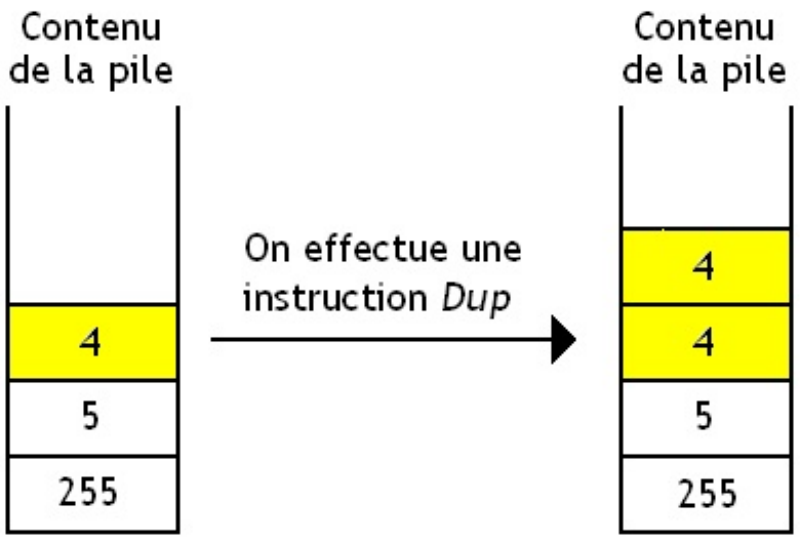
Bien évidemment, on peut aussi ranger la donnée lacée au sommet de la pile dans la mémoire, à une certaine adresse. Dans ce cas, on utilise l'instruction `Pop`, qui dépile la donnée au sommet de la pile et la stocke à l'adresse indiquée dans l'instruction. Encore une fois, le *Stack Pointer* est mis à jour lors de cette opération, en soustrayant (ou additionnant si on fait partir la pile de la fin de la mémoire) la taille de la donnée qu'on vient d'enlever de la pile.



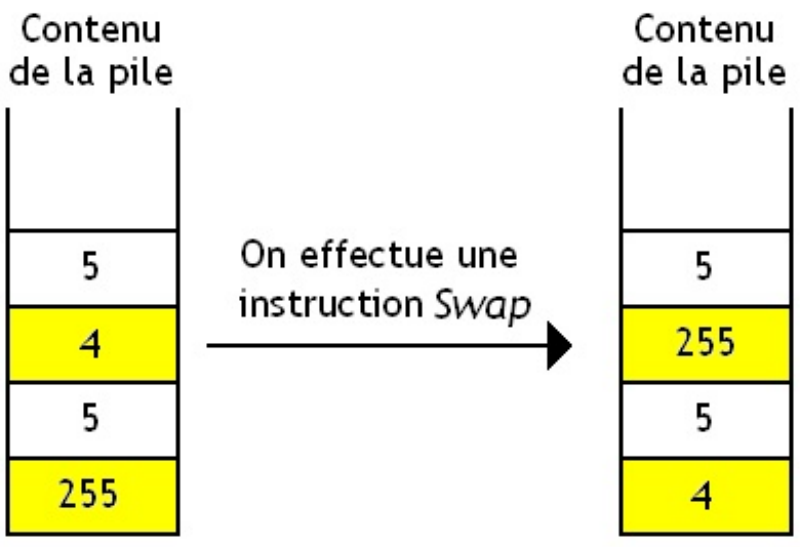
Instructions de traitement de données

Sur une machine à pile, les seules données manipulables par une instruction sont celles qui sont placées au sommet de la pile. Pour exécuter une instruction, il faut donc empiler les opérandes une par une, et exécuter l'instruction une fois que les opérandes sont empilées. Le résultat de l'instruction sera sauvegardé au sommet de la pile.

Chose importante : l'instruction dépile automatiquement les opérandes qu'elle utilise. Elle est un peu obligée, sans quoi la gestion de la pile serait horriblement compliquée, et de nombreuses données s'accumuleraient dans la pile durant un bon moment, faute de pouvoir être dépilées rapidement (vu qu'on empile au-dessus). Ce qui signifie qu'on ne peut pas réutiliser plusieurs fois de suite une donnée placée sur la pile : on doit recharger cette donnée à chaque fois. Ceci dit, certaines instructions ont étés inventées pour limiter la casse. On peut notamment citer l'instruction `dup`, qui copie le sommet de la pile en deux exemplaires.



Pour faciliter la vie des programmeurs, le processeur peut aussi fournir d'autres instructions qui peuvent permettre de manipuler la pile ou de modifier son organisation. On peut par exemple citer l'instruction `swap`, qui échange deux données dans la pile.



Avantages et désavantages

Avec une telle architecture, les programmes utilisent peu de mémoire. Les instructions sont très petites : on n'a pas besoin d'utiliser de bits pour indiquer la localisation des données dans la mémoire, sauf pour `Pop` et `Push`. Vu que les programmes crées pour les machines à pile sont souvent très petits, on dit que la ***code density*** (la densité du code) est bonne. Les machines à pile furent les premières à être inventées et utilisées : dans les débuts de l'informatique, la mémoire était rare et chère, et l'économiser était important. Ces machines à pile permettaient d'économiser de la mémoire facilement, et étaient donc bien vues.

Ces machines n'ont pas besoin d'utiliser beaucoup de registres pour stocker leur état : un *Stack Pointer* et un *Program Counter* suffisent. A peine deux registres (avec éventuellement d'autres registres supplémentaires pour faciliter la conception du processeur).

Par contre, une bonne partie des instructions de notre programmes seront des instructions `Pop` et `Push` qui ne servent qu'à déplacer des données dans la mémoire de notre ordinateur. Une bonne partie des instructions ne sert donc qu'à manipuler la mémoire, et pas à faire des calculs. Sans compter que notre programme comprendra beaucoup d'instructions comparé aux autres types de processeurs.

Machines à accès aléatoire

Ah ben tient, vu qu'on parle de ces autres types de processeurs, voyons ce qu'ils peuvent bien être ! Déjà, pourquoi avoir inventé autre chose que des machines à pile ? Et bien tout simplement pour supprimer les défauts vus plus haut : l'impossibilité de réutiliser une donnée placée sur la pile, et beaucoup de copies ou recopies de données inutiles en mémoire. Pour éviter cela, les concepteurs de processeurs ont inventé des processeurs plus élaborés, qu'on appelle des **machines à accès aléatoire**.

Sur ces ordinateurs, les données qu'une instruction de calcul (une instruction ne faisant pas que lire, écrire, ou déplacer des données dans la mémoire) doit manipuler ne sont pas implicitement placée au sommet d'une pile. Avec les machines à pile, on sait où sont placées ces données, implicitement : le *Stack Pointer* se souvient du sommet de la pile, et on sait alors où sont ces données. Ce n'est plus le cas sur les machines à accès aléatoire : on doit préciser où sont les instructions à manipuler dans la mémoire.

Une instruction doit ainsi fournir ce qu'on appelle une **référence**, qui va permettre de localiser la donnée à manipuler dans la mémoire. Cette référence pourra ainsi préciser plus ou moins explicitement dans quel registre, à quelle adresse mémoire, à quel endroit sur le disque dur, etc ; se situe la donnée à manipuler. Ces références sont souvent stockées directement dans les instructions qui les utilisent, mais on verra cela en temps voulu dans le chapitre sur le langage machine et l'assembleur.

Cela permet d'éviter d'avoir à copier des données dans une pile, les empiler, et les déplacer avant de les manipuler. Le nombre d'accès à la mémoire est plus faible comparé à une machine à pile. Et cela a son importance : il faut savoir qu'il est difficile de créer des mémoires rapides. Et cela devient de plus en plus problématique : de nos jours, le processeur est beaucoup plus rapide que la mémoire. Il n'est donc pas rare que le processeur doive attendre des données en provenance de la mémoire : c'est ce qu'on appelle le *"Von Neumann Bottleneck"*. C'est pour cela que nos ordinateurs actuels sont des machines à accès aléatoire : pour limiter les accès à la mémoire principale.

On peut aussi signaler que quelques anciennes machines et prototypes de recherche ne sont ni des machines à pile, ni des machines à accès aléatoire. Elles fonctionnent autrement, avec des mémoires spéciales : des ***content adressables memory***. Mais

passons : ces architectures sont un peu compliquées, alors autant les passer sur le tapis pour le moment.

Machines à registres

Certaines machines à accès aléatoire assez anciennes ne faisaient que manipuler la mémoire RAM. Les fameuses références mentionnées plus haut étaient donc des adresses mémoires, qui permettaient de préciser la localisation de la donnée à manipuler dans la mémoire principale (la ROM ou la RWM). Pour diminuer encore plus les accès à cette mémoire, les concepteurs d'ordinateurs ont inventés les **machines à registres**.

Ces machines peuvent stocker des données dans des registres intégrés dans le processeur, au lieu de devoir travailler en mémoire. Pour simplifier, ces registres stockent des données comme la pile le faisait sur les machines à pile. Ces registres vont remplacer la pile, mais d'une manière un peu plus souple : on peut accéder à chacun de ces registres individuellement, alors qu'on ne pouvait qu'accéder à une seule donnée avec la pile (celle qui était au sommet de la pile).

Reste à savoir comment charger nos données à manipuler dans ces registres. Après tout, pour la pile, on disposait des instructions `Push` et `Pop`, qui permettaient d'échanger des données entre la pile et la mémoire statique. Sur certains processeurs, on utilise une instruction à tout faire : le `mov`. Sur d'autres, on utilise des instructions séparées suivant le sens de transfert et la localisation des données (dans un registre ou dans la mémoire). Par exemple, on peut avoir des instructions différentes selon qu'on veuille copier une donnée présente en mémoire dans un registre, copier le contenu d'un registre dans un autre, copier le contenu d'un registre dans la mémoire RAM, etc.

Avantages et inconvénients

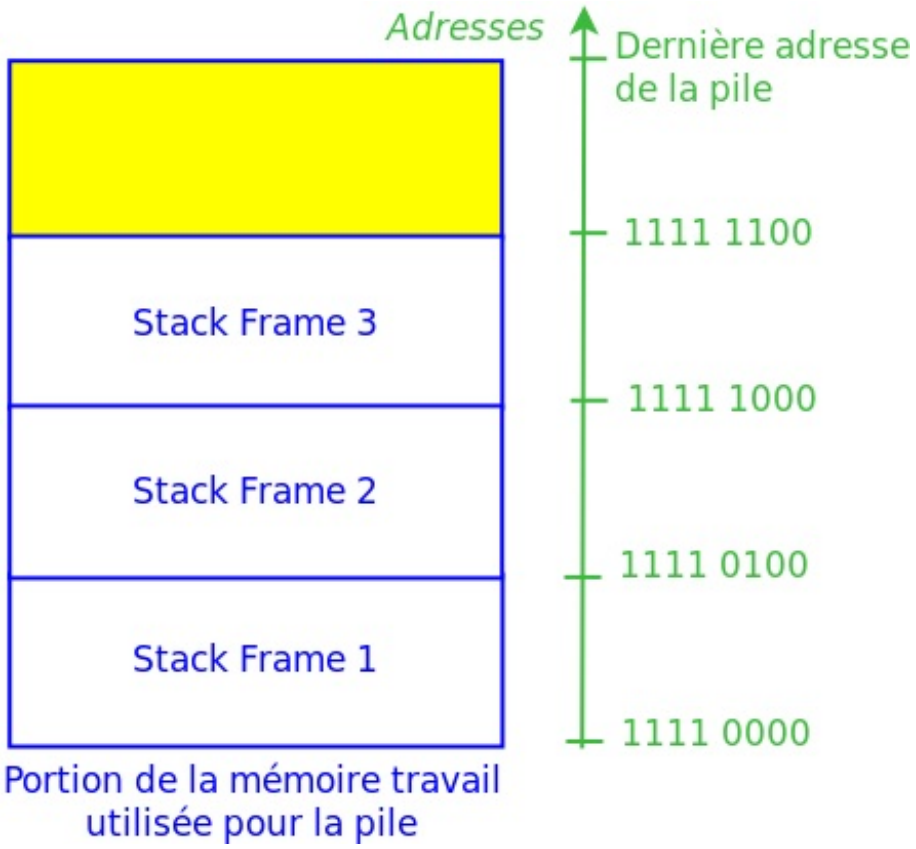
L'utilisation de registres est plus souple que l'utilisation d'une pile. Par exemple, une fois qu'une donnée est chargée dans un registre, on peut la réutiliser autant de fois qu'on veut tant qu'on ne l'a pas effacée. Avec une pile, cette donnée aurait automatiquement effacée, dépilée, après utilisation : on aurait du la recharger plusieurs fois de suite. De manière générale, le nombre total d'accès à la mémoire diminue fortement comparé aux machines à pile.

Et on retrouve les mêmes avantages pour les machines à accès aléatoires n'ayant pas de registres, même si c'est dans une moindre mesure. Il faut dire que nos registres sont souvent des mémoires très rapides, bien plus rapides que la mémoire principale. Utiliser des registres est donc une bonne manière de gagner en performances. C'est pour ces raisons que nos ordinateurs actuels sont souvent des machines à accès aléatoires utilisant des registres.

Le seul problème, c'est qu'il faut bien faire de la place pour stocker les références. Comme je l'ai dit, ces références sont placées dans les instructions : elles doivent préciser où sont stockées les données à manipuler. Et cela prend de la place : des bits sont utilisés pour ces références. La ***code density*** est donc moins bonne. De nos jours, cela ne pose pas vraiment de problèmes : la taille des programmes n'est pas vraiment un sujet de préoccupation majeur, et on peut s'en accommoder facilement.

Les hybrides

De nos jours, on pourrait croire que les machines à accès aléatoire l'ont emporté. Mais la réalité est plus complexe que çà : nos ordinateurs actuels sont certes des machines à accès aléatoire, mais ils possèdent de quoi gérer une pile. Le seul truc, c'est que cette pile n'est pas une pile simple comme celle qui est utilisée sur une machine à pile : la pile de nos ordinateurs utilise des cadres de pile de taille variable. On peut ainsi mettre ce qu'on veut dans ces cadres de pile, et y mélanger des tas de données hétérogènes.



Pour localiser une donnée dans cette *Stack Frame*, il suffit de la repérer en utilisant un décalage par rapport au début ou la fin de celle-ci. Ainsi, on pourra dire : la donnée que je veux manipuler est placée 8 adresses après le début de la *Stack Frame*, ou 16 adresses après la fin de celle-ci. On peut donc calculer l'adresse de la donnée à manipuler en additionnant ce décalage avec le contenu du *Stack Pointer* ou du *Frame Pointer*. Une fois cette adresse connue, nos instructions vont pouvoir manipuler notre donnée en fournissant comme référence cette fameuse adresse calculée.

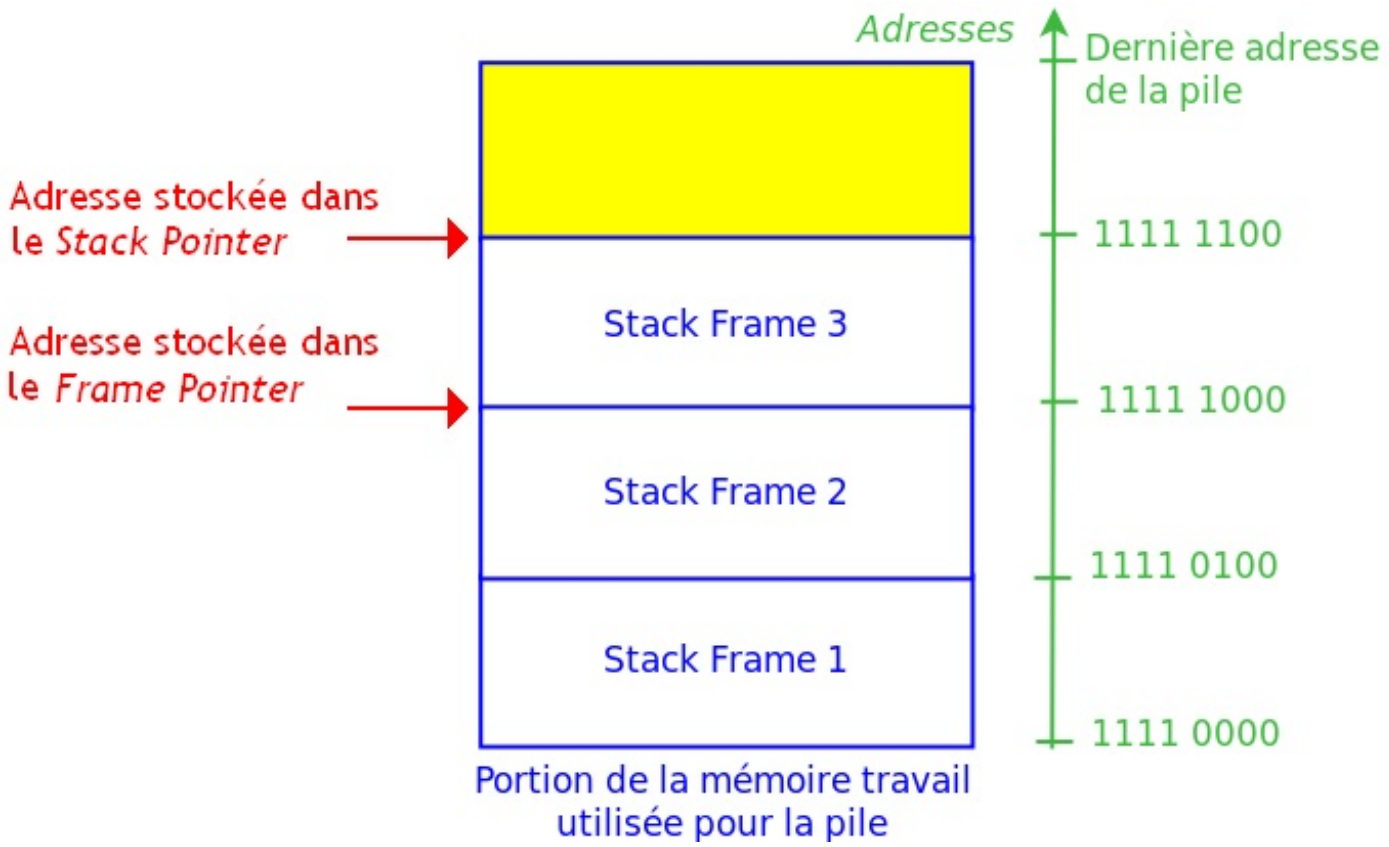
Utiliser des piles aussi compliquées a une utilité : sans cela, certaines fonctionnalités de nos langages de programmation actuels n'existeraient pas ! Pour les connaisseurs, cela signifierait qu'on ne pourrait pas utiliser de fonctions réentrantes ou de fonctions récursives. Mais je n'en dis pas plus : vous verrez ce que cela veut dire d'ici quelques chapitres.

Bon, c'est bien beau, mais ces cadres de pile de taille variables, on les délimite comment ?

Pour cela, on a besoin de sauvegarder deux choses : l'adresse à laquelle commence notre *Stack Frame* en mémoire, et de quoi connaitre l'adresse de fin. Et il existe diverses façons de faire.

Frame Pointer

Pour ce faire, on peut rajouter un registre en plus du *Stack Pointer*, afin de pouvoir gérer ces cadres de pile. Ce registre s'appelle le ***Frame Pointer***, et sert souvent à dire à quelle adresse commence (ou termine, si on fait grandir notre pile de la fin de la mémoire) la *Stack Frame* qui est au sommet de la pile. La création d'une *Stack Frame* se base sur des manipulations de ces deux registres: le *Stack Pointer*, et le *Frame Pointer*.



Certains processeurs possèdent un registre spécialisé qui sert de *Frame Pointer* uniquement : on ne peut pas l'utiliser pour autre chose. Si ce n'est pas le cas, on est obligé de stocker ces informations dans deux registres normaux, et se débrouiller avec les registres restants.

Stack Pointer Only

D'autres processeurs arrivent à se passer de *Frame Pointer*. Ceux-ci n'utilisent pas de registres pour stocker l'adresse de la base de la *Stack Frame*, mais préfèrent calculer cette adresse à partir de l'adresse de fin de la *Stack Frame*, et de sa longueur. Pour info, le calcul est une simple addition/soustraction entre la longueur et le contenu du *Stack Pointer*.

Cette longueur peut être stockée directement dans certaines instructions censées manipuler la pile : si la *Stack Frame* a toujours la même taille, cette solution est clairement la meilleure. Mais il arrive que notre *Stack Frame* aie une taille qui ne soit pas constante : dans ce cas, on a deux solutions : soit stocker cette taille dans un registres, soit la stocker dans les instructions qui manipulent la pile, soit utiliser du *Self Modifying Code* .
Voilà, les bases sont clairement posées : vous avez maintenant un bon aperçu de ce qu'on trouve dans nos ordinateurs. Vous savez ce qu'est un processeur, une mémoire, des bus, et savez plus ou moins comment tout cela est organisé. Vous savez de plus avec quoi sont crées les circuits de notre ordinateurs, surtout pour ce qui est des mémoires. Vous êtes prêt pour la suite.

Dans les chapitres suivants, on va approfondir ces connaissances superficielles, et on va aborder chaque composant (mémoire, processeur, entrées-sorties, bus, etc) uns par uns. Vous saurez comment créer des mémoires complètes, ce qu'il y a dans un processeur, et aurez aussi un aperçu des dernières évolutions technologiques. On peut considérer que c'est maintenant que les choses sérieuses commencent.

Partie 3 : Processeur et Assembleur

C'est maintenant au tour du processeur de passer sur le devant de la scène. Dans cette partie, nous allons commencer par expliquer les bases du langage assembleur, ainsi que du langage machine. Puis, nous descendrons à un niveau plus bas : celui des circuits du processeur et nous finirons par voir comment ceux-ci sont organisés pour concevoir un processeur.

Langage machine et assembleur

Dans ce chapitre, on va aborder **le langage machine d'un processeur**. Le langage machine d'un processeur définit toutes les opérations qu'un programmeur peut effectuer sur notre processeur.
Celui-ci définit notamment :

- les instructions du processeur ;
- ses différentes façons d'accéder à la mémoire ;
- le nombre de registres et leur utilisation ;
- etc.

Au fait : la majorité des concepts qui seront vus dans ce chapitre ne sont rien d'autre que les bases nécessaires pour apprendre l'assembleur. De plus, ce chapitre sera suivi par un chapitre spécialement dédié aux bases théoriques de la programmation en assembleur : boucles, sous-programmes, et autres. C'est sympa, non ?

Instructions

Pour rappel, le rôle d'un processeur est d'exécuter des programmes. Un programme informatique est une suite d'**instructions** à exécuter dans l'ordre. Celles-ci sont placées dans la mémoire programme les unes à la suite des autres dans l'ordre dans lequel elles doivent être exécutées.

C'est quoi une instruction ?

Il existe plusieurs types d'instructions dont voici les principaux :

Instruction	Utilité
Les instructions arithmétiques	Ces instructions font simplement des calculs sur des nombres. On peut citer par exemple : • L'addition ; • la multiplication ; • la division ; • le modulo ; • la soustraction ; • la racine carrée ; • le cosinus ; • et parfois d'autres.
Les instructions logiques	Elles travaillent sur des bits ou des groupes de bits. On peut citer : • Le ET logique. • Le OU logique. • Le XOR. • Le NON , qui inverse tous les bits d'un nombre : les 1 deviennent des 0 et les 0 deviennent des 1. Pour rappel, cela permet de calculer le complément à 1 d'un nombre (rappelez-vous le chapitre sur le binaire) • Les **instructions de décalage** à droite et à gauche, qui vont décaler tous les bits d'un nombre d'un cran vers la gauche ou la droite. Les bits qui sortent du nombre sont considérés comme perdus. • Les **instructions de rotation**, qui font la même chose que les instructions de décalage, à la différence près que les bits qui "sortent d'un côté du nombre" après le décalage rentrent de l'autre.
Les	Certains processeurs intègrent des instructions capables de manipuler ces chaines de caractères

instructions de manipulation de chaines de caractères	directement. Mais autant être franc : ceux-ci sont très rares. Dans notre ordinateur, une lettre est stockée sous la forme d'un nombre souvent codé sur 1 octet (rappelez-vous le premier chapitre sur la table ASCII). Pour stocker du texte, on utilise souvent ce que l'on appelle des **chaines de caractères** : ce ne sont rien de plus que des suites de lettres stockées les unes à la suite des autres dans la mémoire, dans l'ordre dans lesquelles elles sont placées dans le texte.
Les instructions de test	Elles peuvent comparer deux nombres entre eux pour savoir si une condition est remplie ou pas. Pour citer quelques exemples, il existe certaines instructions qui peuvent vérifier si : • deux nombres sont égaux ; • si deux nombres sont différents ; • si un nombre est supérieur à un autre ; • si un nombre est inférieur à un autre.
Les instructions de contrôle	Elles permettent de contrôler la façon dont notre programme s'exécute sur notre ordinateur. Elle permettent notamment de choisir la prochaine instruction à exécuter, histoire de répéter des suites d'instructions, de ne pas exécuter des blocs d'instructions dans certains cas, et bien d'autres choses.
Les instructions d'accès mémoire	Elles permettent d'échanger des données entre le processeur et la mémoire, ou encore permettent de gérer la mémoire et son adressage.
Les instructions de gestion de l'énergie	Elles permettent de modifier la consommation en électricité de l'ordinateur (instructions de mise en veille du PC, par exemple).
Les inclassables	Il existe une grande quantité d'autres instructions, qui sont fournies par certains processeurs pour des besoins spécifiques. • Ainsi, certains processeurs ont des instructions spécialement adaptés aux besoins des OS modernes. • Il arrive aussi qu'on puisse trouver des instructions qui permettent à des programmes de partager des données, d'échanger des informations (via *Message Passing*), etc. etc. • On peut aussi trouver des instructions spécialisées dans les calculs cryptographiques : certaines instructions permettent de chiffrer ou de déchiffrer des données de taille fixe. • De même, certains processeurs ont une instruction permettant de générer des nombres aléatoires. • Certains processeurs sont aussi capables d'effectuer des instructions sur des structures de données assez complexes, comme des listes chainées ou des arbres. • Et on peut trouver bien d'autres exemples...

Ces types d'instructions ne sont pas les seuls : on peut parfaitement trouver d'autres instructions différentes, pour faciliter la création de systèmes d'exploitation, pour manipuler des structures de données plus complexes comme des arbres ou des matrices, etc.

Type des données et instructions

Petite remarque sur les instructions manipulant des nombres (comme les instructions arithmétiques, les décalages, et les tests) : ces instructions dépendent de la représentation utilisée pour ces nombres. La raison est simple : on ne manipule pas de la même façon des nombres signés, des nombres codés en complément à 1, des flottants simple précision, des flottants double précision, etc.

Par exemple, quand on veut faire une addition, on ne traite pas de la même façon un entier ou un flottant. Si vous ne me croyez pas, prenez deux flottants simple précision et additionnez-les comme vous le feriez avec des entiers codés en complément à deux : vous obtiendrez n'importe quoi ! Et c'est pareil pour de nombreuses autres instructions (multiplications, division, etc). On peut

se retrouver avec d'autres cas de ce genre, pour lequel le "type" de la donnée sur laquelle on va instructionner est important.

Dans ce cas, le processeur dispose souvent d'une instruction par type à manipuler. On se retrouve donc avec des instructions différentes pour effectuer la même opération mathématique, chacune de ces instructions étant adaptée à une représentation particulière : on peut avoir une instruction de multiplication pour les flottants, une autre pour les entiers codés en complément à un, une autre pour les entiers codés en *Binary Coded Decimal*, etc.

Sur d'anciennes machines, on stockait le type de la donnée (est-ce un flottant, un entier codé en BCD, etc...) dans la mémoire. Chaque nombre, chaque donnée naturellement manipulée par le processeur incorporait un *tag*, une petite suite de bit qui permettait de préciser son type. Le processeur ne possédait pas d'instruction en plusieurs exemplaires pour faire la même chose, et utilisait le tag pour déduire quoi faire comme manipulation sur notre donnée.

Par exemple, ces processeurs n'avaient qu'une seule instruction d'addition, qui pouvait traiter indifféremment flottants, nombres entiers codés en BCD, en complément à deux, etc. Le traitement effectué par cette instruction dépendait du *tag* incorporé dans la donnée. Des processeurs de ce type s'appellent des ***Tagged Architectures***. De nos jours, ces processeurs n'existent plus que dans quelques muséums : ils ont faits leur temps, laissons-les reposer en paix.

Longueur des données à traiter

La taille des données à manipuler peut elle aussi dépendre de l'instruction. Ainsi, un processeur peut avoir des instructions pour traiter des nombres entiers de 8 bits, et d'autres instructions pour traiter des nombres entiers de 32 bits, par exemple. On peut aussi citer le cas des flottants : il faut bien faire la différence entre flottants simple précision et double précision !

Les tous premiers ordinateurs pouvaient manipuler des données de taille arbitraire : en clair, ils pouvaient manipuler des données aussi grandes qu'on le souhaite sans aucun problème. Alors certes, ces processeurs utilisaient des ruses : ils n'utilisait pas vraiment le binaire qu'on a vu au premier chapitre.

A la place, ils stockaient leurs nombres dans des chaines de caractères ou des tableaux encodés en *Binary Coded Decimal* (une méthode de représentation des entiers assez proche du décimal), et utilisaient des instructions pouvant manipuler de tels tableaux. Mais de nos jours, cela tend à disparaitre, et les processeurs ne disposent plus d'instructions de ce genre.

Jeux d'instruction

Au fait, on va mettre les choses au clair tout de suite : certains processeurs peuvent faire des instructions que d'autres ne peuvent pas faire. Ainsi, les instructions exécutables par un processeur dépendent fortement du processeur utilisé. La liste de toute les instructions qu'un processeur peut exécuter s'appelle son **jeu d'instruction**. Ce jeu d'instruction va définir quelles sont les instructions supportées, ainsi que les suites de bits correspondant à chaque instruction.

RISC vs CISC

Il existe différents jeux d'instructions : le X86 , le PPC, etc. Et tout ces jeux d'instructions ont leurs particularités. Pour s'y retrouver, on a grossièrement classé ces jeux d'instructions en plusieurs catégories. La première classification se base sur le nombre d'instructions et classe nos processeurs en deux catégories :

- les **RISC** (reduced instruction set computer) qui peuvent exécuter très peu d'instructions ;
- et les **CISC** (complex instruction set computer) avec pleins d'instructions.

CISC

CISC est l'acronyme de ***Complex Instruction Set Computer***. Traduit de l'anglais cela signifie Ordinateur à jeu d'instruction complexe. Les processeurs CISC ont un jeu d'instruction étoffé, avec beaucoup d'instructions. De plus, certaines de ces instructions sont assez complexes et permettent de faire des opérations assez évoluées.

Par exemple, ces processeurs peuvent :

- calculer des fonctions mathématiques compliquées comme des sinus, cosinus, logarithmes, etc ;
- ont des instructions pour traiter du texte directement ;
- peuvent gérer des structures de données complexes, comme des tableaux ;
- etc.

Ces jeux d'instructions sont les plus anciens : ils étaient à la mode jusqu'à la fin des années 1980. A cette époque, on programmait rarement avec des langages de haut niveau et beaucoup de programmeurs devaient utiliser l'assembleur. Avoir un jeu d'instruction complexe, avec des instructions de "haut niveau" qu'on ne devait pas refaire à partir d'instructions plus simples, était un gros avantage : cela facilitait la vie des programmeurs.

Cette complexité des jeux d'instructions n'a pas que des avantages "humains", mais a aussi quelques avantages techniques. Il n'est pas rare qu'une grosse instruction complexe puisse remplacer une suite d'instructions plus élémentaires.

Cela a quelques effets plutôt bénéfiques :

- une grosse instruction lente peut être plus rapide à exécuter que plusieurs instructions rapides ;
- un programme écrit pour un processeur CISC comportera moins d'instructions.

Vu qu'un programme écrit pour des processeurs CISC utilise moins d'instructions, il prendra donc moins de place en mémoire programme. A l'époque des processeurs CISC, la mémoire était rare et chère, ce qui faisait que les ordinateurs n'avaient pas plusieurs gigaoctets de mémoire : économiser celle-ci était crucial.

Mais ces avantages ne sont pas sans contreparties :

- on a besoin de beaucoup de circuits pour câbler toutes ces instructions ;
- ces circuits ont tendance à chauffer ;
- ces circuits consomment de l'énergie ;
- le processeur est plus compliqué à concevoir ;
- etc.

L'agence tout RISC

Au fil du temps, on s'est demandé si les instructions complexes des processeurs CISC étaient vraiment utiles. Pour le programmeur qui écrit ses programmes en assembleur, elle le sont. Mais avec l'invention des langages de haut niveau, la roue a commencée à tourner. Diverses analyses ont alors étés effectuées par IBM, DEC et quelques chercheurs, visant à évaluer les instructions réellement utilisées par les compilateurs. Et à l'époque, les compilateurs n'utilisaient pas la totalité des instructions fournies par un processeur. Nombre de ces instructions n'étaient utilisées que dans de rares cas, voire jamais. Autant dire que beaucoup de transistors étaient gâchés à rien !

L'idée de créer des processeurs possédant des jeux d'instructions simples et contenant un nombre limité d'instructions très rapides commença à germer. Ces processeurs sont de nos jours appelés des processeurs RISC. RISC est l'acronyme de *Reduced Instruction Set Computer*. Traduit de l'anglais cela signifie Ordinateur à jeu d'instruction réduit.

Mais de tels processeurs RISC, complètement opposés aux processeurs CISC, durent attendre un peu avant de percer. Par exemple, IBM décida de créer un processeur possédant un jeu d'instruction plus sobre, l'IBM 801, qui fût un véritable échec commercial. Mais la relève ne se fit pas attendre. C'est dans les années 1980 que les processeurs possédant un jeu d'instruction simple devinrent à la mode. Cette année là, un scientifique de l'université de Berkeley décida de créer un processeur possédant un jeu d'instruction contenant seulement un nombre réduit d'instructions simples, possédant une architecture particulière. Ce processeur était assez novateur et incorporait de nombreuses améliorations qu'on retrouve encore dans nos processeurs haute performances actuels, ce qui fit son succès : les processeurs RISC étaient nés.

Comme ce qui a été dit plus haut, un processeur RISC n'a pas besoin de cabler beaucoup d'instructions, ce qui a certains effets assez bénéfiques :

- un processeur RISC utilise peu de circuits électroniques ;
- il est donc souvent plus simple à concevoir ;
- il chauffe moins ;
- il consomme moins d'énergie ;
- il est plus simple à utiliser pour un compilateur ;

Mais par contre, cela a aussi quelques désavantages :

- les programmes compilés sur les processeurs RISC prennent plus de mémoire ;
- certaines instructions complexes qui permettaient de gagner en performances ne sont pas disponibles, ce qui signifie une perte en performance ;
- ils sont beaucoup compliqués à programmer en assembleur.

Qui est le vainqueur ?

Durant longtemps, les CISC et les RISC eurent chacun leurs admirateurs et leurs détracteurs. De longs et interminables débats eurent lieu pour savoir si les CISC étaient meilleurs que les RISC, similaires aux "Windows versus Linux", ou "C versus C++", qu'on trouve sur n'importe quel forum digne de ce nom. Au final, on ne peut pas dire qu'un processeur CISC sera meilleur qu'un

RISC ou l'inverse : chacun a des avantages et des inconvénients, qui rendent le RISC/CISC adapté ou pas selon la situation.

Par exemple, on mettra souvent un processeur RISC dans un système embarqué, devant consommer très peu. Par contre, le CISC semble mieux adapté dans certaines conditions, en raison de la taille plus faible des programmes, ou quand les programmes peuvent faire un bon usage des instructions complexes du processeur.

Au final, tout dépend d'un tas de paramètres :

- suivant les besoins du programme à exécuter ;
- le langage de programmation utilisé ;
- la qualité du compilateur ;
- la façon dont est conçue le processeur ;
- les instructions disponibles ;
- les spécificités du jeu d'instruction ;
- l'age du capitaine.

Tout ces paramètres jouent beaucoup dans la façon dont on pourra tirer au mieux parti d'un processeur RISC ou CISC, et ils sont bien plus importants que le fait que le processeur soit un RISC ou un CISC.

De plus, de nos jours, les différences entre CISC et RISC commencent à s'estomper. Les processeurs actuels sont de plus en plus difficiles à ranger dans des catégories précises. Les processeurs actuels sont conçus d'une façon plus pragmatiques : au lieu de respecter à la lettre les principes du RISC et du CISC, on préfère intégrer les techniques et instructions qui fonctionnent, peut importe qu'elles viennent de processeurs purement RISC ou CISC. Les anciens processeurs RISC se sont ainsi garnis d'instructions et techniques de plus en plus complexes et les processeurs CISC ont intégré des techniques provenant des processeurs RISC (pipeline, etc). Au final, cette guerre RISC ou CISC n'a plus vraiment de sens de nos jours.

Jeux d'instructions spécialisés

En parallèle de ces architectures CISC et RISC, qui sont en quelques sorte la base de tous les jeux d'instructions, d'autres classes de jeux d'instructions sont apparus, assez différents des jeux d'instructions RISC et CISC. On peut par exemple citer le ***Very Long Instruction Word***, qui sera abordé dans les chapitre à la fin du tutoriel. La plupart de ces jeux d'instructions sont implantés dans des processeurs spécialisés, qu'on fabrique pour une utilisation particulière. Ce peut être pour un langage de programmation particulier, pour des applications destinées à un marche de niche comme les supercalculateurs, etc.

Les DSP

Parmi ces jeux d'instructions spécialisés, on peut citer les fameux jeux d'instructions ***Digital Signal Processor***, aussi appelés des ***DSP***. Ces DSP sont des processeurs chargés de faire des calculs sur de la vidéo, du son, ou tout autre signal. Dès que vous avez besoin de traiter du son ou de la vidéo, vous avez un DSP quelque part, que ce soit une carte son ou une platine DVD.

Ces DSP ont souvent un jeu d'instruction similaire aux jeux d'instructions RISC, avec peu d'instructions, toutes spécialisées pour faire du traitement de signal. On peut par exemple citer l'instruction phare de ces DSP, l'instruction `MAD` (qui multiplie deux nombres et additionne un 3éme au résultat de la multiplication). De nombreux algorithmes de traitement du signal (filtres FIR, transformées de Fourier) utilisent massivement cette opération. Ces DSP possèdent aussi des instructions permettant de faire répéter rapidement une suite d'instruction (pour les connaisseurs, ces instructions permettent de créer des boucles), ou des instructions capables de traiter plusieurs données en parallèle (en même temps).

Ces instructions manipulent le plus souvent des nombres entiers, et parfois (plus rarement) des nombres flottants. Ceci dit, ces DSP utilisent souvent des nombres flottants assez particuliers qui n'ont rien à voir avec les nombres flottants que l'on a vu dans le premier chapitre. Il supportent aussi des formats de nombre entiers assez exotiques, même si c'est assez rare.

Ces DSP ont souvent une architecture de type Harvard. Pour rappel, cela signifie qu'ils sont connectés à deux bus de données : un pour les instructions du programme, et un autre relié à la mémoire de travail, pour les données. Certains DSP vont même plus loin : ils sont reliés à plusieurs bus mémoire. Au bout de ces bus mémoire, on retrouve souvent plusieurs mémoires séparées. Nos DSP sont donc capables de lire et/ou d'écrire plusieurs données simultanément : une par bus mémoire relié au DSP.

Il y a pire

On peut aussi citer les jeux d'instructions de certains processeurs particulièrement adaptés à un système d'exploitation en particulier. Un exemple serait les processeurs *multics*, spécialement dédiés au système d'exploitation du même nom. Il faut avouer que ces processeurs sont assez rares et dédiés à des marchés de niche.

Dans le même genre, certains processeurs sont spécialement conçus pour un langage en particulier. Il existe ainsi des processeurs possédant des instructions permettant d'accélérer le traitement des opérations de base fournies par un langage de

programmation, ou encore d'implémenter celle-ci directement dans le jeu d'instruction du processeur, transformant ainsi ce langage de haut niveau en assembleur. On appelle de tels processeurs, des **processeurs dédiés**.

Historiquement, les premiers processeurs de ce type étaient des processeurs dédiés au langage LISP, un vieux langage fonctionnel autrefois utilisé, mais aujourd'hui peu usité. De tels processeurs datent des années 1970 et étaient utilisés dans ce qu'on appelait des machines LISP. Ces machines LISP étaient capables d'exécuter certaines fonctions de base du langage directement dans leur circuits : elles possédaient notamment un *garbage collector* câblé dans ses circuits ainsi que des instructions machines supportant un typage déterminé à l'exécution. D'autres langages fonctionnels ont aussi eu droit à leurs processeurs dédiés : le prolog en est un bel exemple.

Autre langage qui a eu l'honneur d'avoir ses propres processeurs dédiés : le FORTH, un des premiers langages à pile de haut niveau. Ce langage possède de nombreuses implémentations hardware et est un des rares langages de haut niveau à avoir été directement câblé en assembleur sur certains processeurs. Par exemple, on peut citer le processeur FC16, capable d'exécuter nativement du FORTH.

En regardant dans les langages de programmation un peu plus connus, on peut aussi citer des processeurs spécialisés pour JAVA, qui intègrent une machine virtuelle JAVA directement dans leurs circuits : de quoi exécuter nativement du *bytecode* ! Certains processeurs ARM, qu'on trouve dans des système embarqués, sont de ce type.

Et pour nos ordinateurs ?

Le jeu d'instruction de nos PC qui fonctionnent sous Windows est appelé le **x86**. C'est un jeu d'instructions particulièrement ancien, apparu certainement avant votre naissance : 1978. Depuis, de plus en plus d'instructions ont été ajoutées et rajoutées : ces instructions sont ce qu'on appelle des **extensions x86**. On peut citer par exemple les extensions MMX, SSE, SSE2, voir 3dnow!. Le résultat, c'est que les processeurs x86 sont de type CISC, avec tous les inconvénients que cela apporte.

Les anciens macintoshs (la génération de macintosh produits entre 1994 et 2006) utilisaient un jeu d'instruction différent : le **PowerPC**. Celui-ci était un jeu d'instruction de type RISC. Depuis 2006, les macintoshs utilisent un processeur X86.

Mais les architectures x86 et Power PC ne sont pas les seules au monde : il existe d'autres types d'architectures qui sont très utilisées dans le monde de l'informatique embarquée et dans tout ce qui est tablettes et téléphones portables derniers cris. On peut citer notamment l'architecture ARM, qui domine ce marché. Et n'oublions pas leurs consœurs MIPS et SPARC.

Registres architecturaux

Nos instructions manipulent donc des données, qui sont forcément stockées quelques part dans la mémoire de notre ordinateur. En plus d'avoir accès aux données placées dans la mémoire RAM, le processeur possède plusieurs mémoires internes très rapides qui peuvent stocker très peu de données : des **registres.** Ces registres servent à stocker temporairement des informations dont le processeur peut avoir besoin, aussi bien instructions, adresses ou données. Il s'agit bien des registres vus dans les chapitres précédents, fabriqués avec des bascules.

Mais pourquoi utiliser des registres pour stocker des données alors que l'on a déjà une mémoire RAM ?

C'est très simple : la mémoire RAM est une mémoire assez lente, et y accéder en permanence rendrait notre ordinateur vraiment trop lent pour être utilisable. Par contre, les registres sont des mémoires vraiment très rapides. En stockant temporairement des données dans ces registres, le processeur pourra alors manipuler celle-ci très rapidement, sans avoir à attendre une mémoire RAM à la lenteur pachydermique. Typiquement, dès qu'une donnée doit être lue ou modifiée plusieurs fois de suite, on a tout à gagner à la mettre dans un registre.

A quoi servent ces registres ?

On peut se demander à quoi servent ces registres. Tout cela dépend du processeur, et tous nos processeurs ne gèrent pas ces registres de la même façon.

Registres spécialisés

Certains processeurs disposent de registres spécialisés, qui ont une utilité bien précise. Leur fonction est ainsi prédéterminée une bonne fois pour toute. Le contenu de nos registres est aussi fixé une bonne fois pour toute : un registre est conçu pour stocker soit des nombres entiers, des flottants, des adresses, etc; mais pas autre chose. Pour donner quelques exemples, voici quelques registres spécialisés qu'on peut trouver sur pas mal de processeurs.

Registre	Utilité

Le registre d'adresse d'instruction	Pour rappel, un processeur doit effectuer une suite d'instructions dans un ordre bien précis. Dans ces conditions, il est évident que notre processeur doit se souvenir où il est dans le programme, quelle est la prochaine instruction à exécuter : notre processeur doit donc contenir une mémoire qui stocke cette information. C'est le rôle du registre d'adresse d'instruction. Ce registre stocke l'adresse de la prochaine instruction à exécuter. Cette adresse permet de localiser l'instruction suivante en mémoire. Cette adresse ne sort pas de nulle part : on peut la déduire de l'adresse de l'instruction en cours d'exécution par divers moyens plus ou moins simples qu'on verra dans la suite de ce tutoriel. Cela peut aller d'une simple addition à quelque chose d'un tout petit peu plus complexe. Quoiqu'il en soit, elle est calculée par un petit circuit combinatoire couplé à notre registre d'adresse d'instruction, qu'on appelle le compteur ordinal. Ce registre d'adresse d'instruction est souvent appelé le ***Program Counter***. Retenez bien ce terme, et ne l'oubliez pas si vous voulez lire des documentations en anglais.
Le registre d'état	Le registre d'état contient plusieurs bits qui ont chacun une utilité particulière. Ce registre est très différent suivant les processeurs, mais certains bits reviennent souvent : • **divers bits utilisés lors d'opérations de comparaisons ou de tests** qui servent à donner le résultat de celles-ci ; • le **bit d'overflow**, qui prévient quand le résultat d'une instruction est trop grand pour tenir dans un registre ; • le **bit null** : précise que le résultat d'une instruction est nul (vaut zéro) ; • le **bit de retenue**, utile pour les additions ; • le **bit de signe**, qui permet de dire si le résultat d'une instruction est un nombres négatif ou positif.
Le *Stack Pointer*, et éventuellement le *Frame Pointer*	Ces deux registres sont utilisés pour gérer une pile, si le processeur en possède une. Pour ceux qui auraient oubliés ce qu'est la pile, le chapitre 5 est là pour vous. Pour rappel, le *Stack Pointer* stocke l'adresse du sommet de la pile. Tout processeur qui possède une pile en possède un. Par contre, le *Frame Pointer* est optionnel : il n'est présent que sur les processeurs qui gèrent des *Stack Frames* de taille variable. Ce registre stocke l'adresse à laquelle commence la *Stack Frame* située au sommet de la pile.
Registres entiers	Certains registres sont spécialement conçus pour stocker des nombres entiers. On peut ainsi effectuer des instructions de calculs, des opérations logiques dessus.
Registres flottants	Certains registres sont spécialement conçus pour stocker des nombres flottants. L'intérêt de placer les nombres flottants à part des nombres entiers, dans des registres différents peut se justifier par une remarque très simple : on ne calcule pas de la même façon avec des nombres flottants et avec des nombres entiers. La façon de gérer les nombres flottants par nos instructions étant différente de celle des entiers, certains processeurs placent les nombres flottants à part, dans des registres séparés. On peut ainsi effectuer des instructions de calculs, des opérations logiques dessus.
Registres de constante	Ces registres de constante contiennent des constantes assez souvent utilisées. Par exemple, certains processeurs possèdent des registres initialisés à zéro pour accélérer la comparaison avec zéro ou l'initialisation d'une variable à zéro. On peut aussi citer certains registres flottants qui stockent des nombres comme π, ou e pour faciliter l'implémentation des calculs trigonométriques).
Registres d'Index	Autrefois, nos processeurs possédaient des registres d'*Index*, qui servait à calculer des adresses, afin de manipuler rapidement des données complexes comme les tableaux. Ces registres d'*Index* étaient utilisés pour effectuer des manipulations arithmétiques sur des adresses. Sans eux, accéder à des données placées à des adresses mémoires consécutives nécessitait souvent d'utiliser du self-modifying code : le programme devait être conçu pour se modifier lui-même en partie, ce qui n'était pas forcément idéal pour le programmeur.

Sur certains processeurs, certains des registres cités plus bas ne sont pas présents ! Il existe ainsi des processeurs qui se passent des registres chargés de gérer la pile : tout processeur n'utilisant pas de pile le peut. De même, il est possible de se passer du registre d'état, etc.

Registres généraux

Malheureusement, fournir des registres très spécialisés n'est pas très flexible. Prenons un exemple : j'ai un processeur disposant d'un *Program Counter*, de 4 registres entiers, de 4 registres d'*Index* pour calculer des adresses, et de 4 registres flottants. Si jamais j'exécute un morceau de programme qui manipule beaucoup de nombres entiers, mais qui ne manipule pas d'adresses ou de nombre flottants, j'utiliserais juste les 4 registres entiers. Une partie des registres du processeur sera inutilisé : tous les registres flottants et d'*Index*. Le problème vient juste du fait que ces registres ont une fonction bien fixée.

Pourtant, en réfléchissant, un registre est un registre, et il ne fait que stocker une suite de bits. Il peut tout stocker : adresses, flottants, entiers, etc. Pour plus de flexibilité, certains processeurs ne fournissent pas de registres spécialisés comme des registres entiers ou flottants, mais fournissent à la place des **Les registres généraux** utilisables pour tout et n'importe quoi. Ce sont des registres qui n'ont pas d'utilité particulière et qui peuvent stocker toute sorte d'information codée en binaire. Pour reprendre notre exemple du dessus, un processeur avec des registres généraux fournira un *Program Counter* et 12 registres généraux, qu'on peut utiliser sans vraiment de restrictions. On pourra s'en servir pour stocker 12 entiers, 10 entiers et 2 flottants, 7 adresses et 5 entiers, etc. Ce qui sera plus flexible et permettra de mieux utiliser les registres.

Dans la réalité, nos processeurs utilisent souvent un espèce de mélange entre les deux solutions. Généralement, une bonne partie des registres du processeur sont des registres généraux, à part quelques registres spécialisés, accessibles seulement à travers quelques instructions bien choisies. C'est le cas du registre d'adresse d'instruction, qui est manipulé automatiquement par le processeur et par les instructions de branchement.

La catastrophe

Ceci dit, certains processeurs sont très laxistes : tous les registres sont des registres généraux, même le *Program Counter*. Sur ces processeurs, on peut parfaitement lire ou écrire dans le *Program Counter* sans trop de problèmes. Ainsi, au lieu d'effectuer des branchements sur notre *Program Counter*, on peut simplement utiliser une instruction qui ira écrire l'adresse à laquelle brancher dans notre registre. On peut même faire des calculs sur le contenu du *Program Counter* : cela n'a pas toujours de sens, mais cela permet parfois d'implémenter facilement certains types de branchements avec des instructions arithmétiques usuelles.

Registres architecturaux

Un programmeur (ou un compilateur) qui souhaite programmer en langage machine peut manipuler ces registres. A ce stade, il faut faire une petite remarque : tous les registres d'un processeur ne sont pas forcément manipulables par le programmeur. Il existe ainsi deux types de registres : les **registres architecturaux**, manipulables par des instructions, et d'autres registres internes au processeurs. Ces registres peuvent servir à simplifier la conception du processeur ou à permettre l'implémentation d'optimisations permettant de rendre notre ordinateur plus rapide.

Le nombre de registres architecturaux varie suivant le processeur. Généralement, les processeurs RISC et les DSP possèdent un grand nombre de registres. Sur les processeurs CISC, c'est l'inverse : il est rare d'avoir un grand nombre de registres architecturaux manipulables par un programme. Quoiqu'il en soit, tous les registres cités plus haut sont des registres architecturaux.

Ça doit être du sport pour se retrouver dans un processeur avec tout ces registres ! Comment notre programmeur fait-il pour sélectionner un registre parmi tous les autres ?

Et bien rassurez-vous, les concepteurs de processeurs ont trouvé des solutions.

Registres non référencables

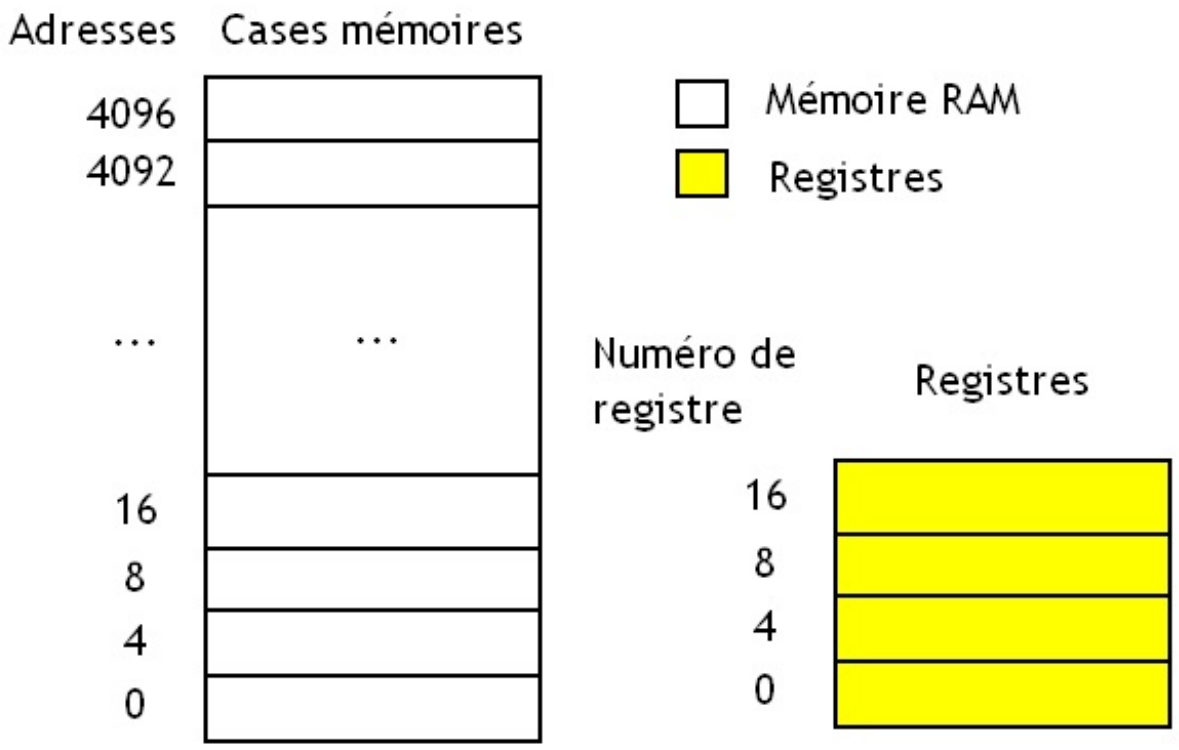
Certains registres n'ont pas besoin d'être sélectionnées. On les manipule implicitement avec certaines instructions. Le seul moyen de manipuler ces registres est de passer par une instruction appropriée, qui fera ce qu'il faut. C'est le cas pour le *Program Counter* : à part sur certains processeurs vraiment très rares, on ne peut modifier son contenu qu'en utilisant des instructions de branchements. Idem pour le registre d'état, manipulé implicitement par les instructions de comparaisons et de test, et certaines opérations arithmétiques.

Noms de registres

Dans le premier cas, chaque registre se voit attribuer une référence, une sorte d'identifiant qui permettra de le sélectionner parmi tous les autres. C'est un peu la même chose que pour la mémoire RAM : chaque *byte* de la mémoire RAM se voit attribuer une adresse bien précise. Et bien pour les registres, c'est un peu la même chose : ils se voient attribuer quelque chose d'équivalent à une adresse, une sorte d'identifiant qui permettra de sélectionner un registre pour y accéder.

Cet identifiant est ce qu'on appelle un **nom de registre**. Ce nom n'est rien d'autre qu'une suite de bits attribuée à chaque registre, chaque registre se voyant attribuer une suite de bits différente. Celle-ci sera intégrée à toutes les instructions devant manipuler

ce registre, afin de sélectionner celui-ci. Ce numéro, ou nom de registre, permet d'identifier le registre que l'on veut, mais ne sort jamais du processeur : ce nom de registre, ce numéro, ne se retrouve jamais sur le bus d'adresse. Les registres ne sont donc pas identifiés par une adresse mémoire.

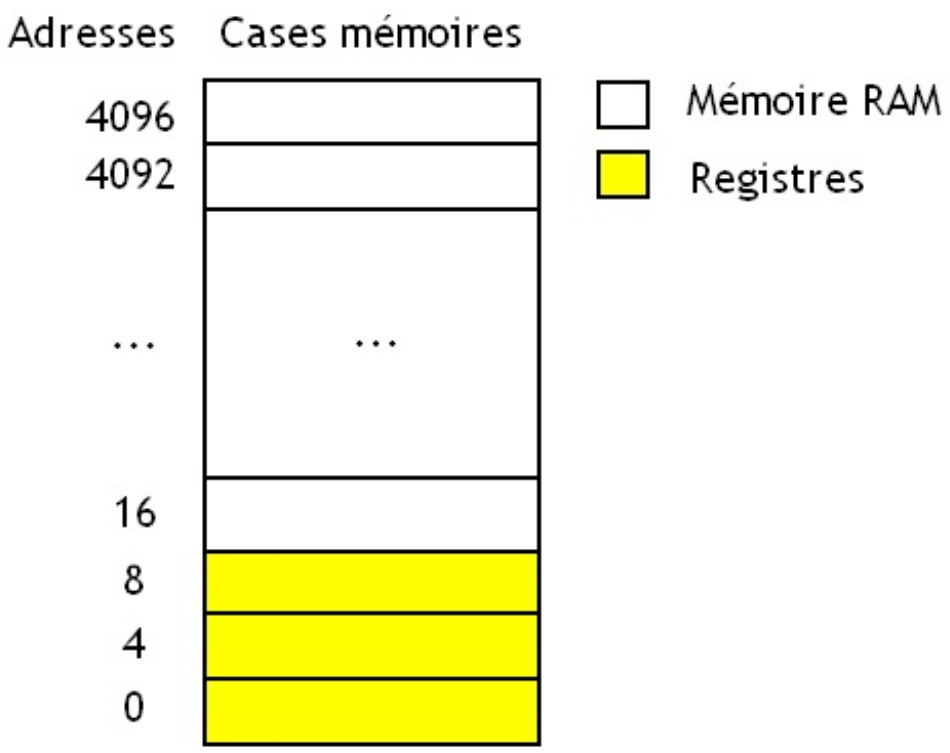


Toutefois, tous les registres n'ont pas forcément besoin d'avoir un nom. Par exemple, les registres chargés de gérer la pile n'ont pas forcément besoin d'un nom : la gestion de la pile se fait alors via des instructions `Push` et `Pop` qui sont les seules à pouvoir manipuler ces registres. Toute manipulation du *Frame Pointer* et du *Stack Pointer* se faisant grâce à ces instructions, on n'a pas besoin de leur fournir un identifiant pour pouvoir les sélectionner. C'est aussi le cas du registre d'adresse d'instruction : sur certains processeurs, il est manipulé automatiquement par le processeur et par les instructions de branchement. Dans ces cas bien précis, on n'a pas besoin de préciser le ou les registres à manipuler : le processeur sait déjà quels registres manipuler et comment, de façon implicite. Quand on effectue un branchement, le processeur sait qu'il doit modifier le *Program Counter* : pas besoin de lui dire. Pareil pour les instructions de gestion de la pile.

Ces noms de registres posent un petit problème. Quand une instruction voudra manipuler des données, elle devra fatalement donner une adresse ou un nom de registres qui indiquera la position de la donnée en mémoire. Ces adresses et noms de registres sont codés sous la forme de suites de bits, incorporées dans l'instruction. Mais rien ne ressemble plus à une suite de bits qu'une autre suite de bits : notre processeur devra éviter de confondre suite de bits représentant une adresse, et suite de bits représentant un nom de registre. Pour éviter les confusions, chaque instruction devra préciser à quoi correspondra la suite de bits précisant la localisation des données à manipuler : est-ce un registres ou une adresse, ou autre chose encore. Cette précision (cet-ce une adresse ou un nom de registres) sera indiquée par ce qu'on appelle un **mode d'adressage**. Nous reviendront dessus tout à l'heure.

Registres adressables

Mais il existe une autre solution, assez peu utilisée. Sur certains processeurs assez rares, on peut adresser les registres via une adresse mémoire. Il est vrai que c'est assez rare, et qu'à part quelques vielles architectures ou quelques micro-contrôleurs, je n'ai pas d'exemples à donner. Mais c'est tout à fait possible ! C'est le cas du PDP-10.



8, 16, 32, 64 bits : une histoire de taille des registres

Vous avez déjà entendu parler de processeurs 32 ou 64 bits ?

Derrière cette appellation qu'on retrouve souvent dans la presse ou comme argument commercial se cache un concept simple. Il s'agit de la quantité de bits qui peuvent être stockés dans chaque registre généraux.

Attention : on parle bien des registres généraux, et pas forcément des autres registres. Notre processeur contient pas mal de registres et certains peuvent contenir plus de bits que d'autres. Par exemple, dans certains processeurs, les registres généraux sont séparés des registres stockant des flottants et ces deux types de registres peuvent avoir une taille différente. Exemple : dans les processeurs x86, il existe des registres spécialement dédiés aux nombres flottants et d'autres spécialement dédiés aux nombres entiers (ce sont les registres généraux qui servent pour les entiers). Les registres pour nombres entiers n'ont pas la même taille que les registres dédiés aux nombres flottants. Un registre pour les nombres entiers contient environ 32 bits tandis qu'un registre pour nombres flottants contient 80 bits.

Ce nombre de bits que peut contenir un registre est parfois différent du nombre de bits qui peuvent transiter en même temps sur le bus de donnée de votre ordinateur. Cette quantité peut varier suivant l'ordinateur. On l'appelle la **largeur du bus de données**. Exemple : sur les processeurs x 86 - 32 bits, un registre stockant un entier fait 32bits. Un registre pour les flottants en fait généralement 64. Le bus de donnée de ce genre d'ordinateur peut contenir 64 bits en même temps. Cela a une petite incidence sur la façon dont une donnée est transférée entre la mémoire et un registre. On peut donc se retrouver dans deux situations différentes :

Situation	Conséquence
Le bus de données a une largeur égale à la taille d'un registre	Le bus de donnée peut charger en une seule fois le nombre de bits que peut contenir un registre.
La largeur du bus de donnée est plus petite que la taille d'un registre	On ne peut pas charger le contenu d'un registre en une fois, et on doit charger ce contenu morceau par morceau.

Représentation en binaire

On peut de demander comment notre ordinateur fait pour stocker ces instructions dans sa mémoire. On a déjà vu il y a quelques chapitres que les instructions sont stockées dans la mémoire programme de l'ordinateur sous la forme de suites de bits.

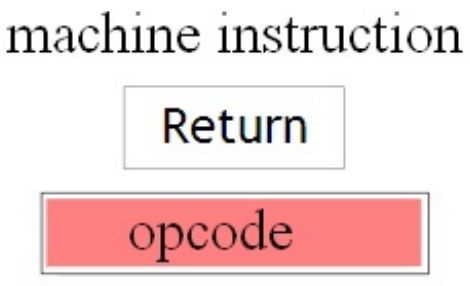
Exemple : *ici, les valeurs binaires sont complètement fictives.*

Instruction	Valeur Binaire
Ne rien faire durant un cycle d'horloge : NOP	1001 0000
Mise en veille : HALT	0110 1111
Addition : ADD	0000 0000, ou 0000 0001, ou 1000 0000, etc...
Écriture en mémoire : STORE	1111 1100, ou 1111 1101, ou 1111 1110, etc...

Mais j'ai volontairement passé sous silence quelque chose : cette suite de bits n'est pas organisée n'importe comment.

Opcode

La suite de bits de notre instruction contient une portion qui permet d'identifier l'instruction en question. Cette partie permet ainsi de dire s'il s'agit d'une instruction d'addition, de soustraction, d'un branchement inconditionnel, d'un appel de fonction, d'une lecture en mémoire, etc. Cette portion de mémoire s'appelle l'**opcode**.



Pour la même instruction, l'opcode peut être différent suivant le processeur, ce qui est source d'incompatibilité. Ce qui fait que pour chaque processeur, ses fabricants donnent une liste qui recense l'intégralité des instructions et de leur opcode : **l'opcode map**.

Petit détail : il existe certains processeurs qui utilisent une seule et unique instruction. Ces processeurs peuvent donc se passer d'opcode : avec une seule instruction possible, pas besoin d'avoir un opcode pour préciser quelle instruction exécuter. Mais autant prévenir : ces processeurs sont totalement tordus et sont vraiment très rares. Inutile de s'attarder plus longtemps sur ces processeurs.

Opérandes

Il arrive que certaines instructions soient composées d'un Opcode, sans rien d'autre. Elles ont alors une représentation en binaire qui est unique. Mais certaines instructions ne se contentent pas d'un opcode : elles utilisent une partie variable. Cette partie variable peut permettre de donner des informations au processeur sur l'instruction, sur ses données, ou permettre d'autres choses encore. Mais le plus fréquemment, cette partie variable permet de préciser quelles sont les données à manipuler. Sans cela, rien ne marche !

Quand je dis "préciser quelles sont les données à manipuler", cela veut vouloir dire plusieurs choses. On peut parfois mettre la donnée directement dans l'instruction : si la donnée est une constante, on peut la placer directement dans l'instruction. Mais dans les autres cas, notre instruction va devoir préciser la localisation des données à manipuler : est-ce que la donnée à manipuler est dans un registre (et si oui, lequel), dans la mémoire (et à quelle adresse ?). De même, où enregistrer le résultat ? Bref, cette partie variable est bien remplie.

Modes d'adressage

Reste à savoir comment interpréter cette partie variable : après tout, c'est une simple suite de bits qui peut représenter une adresse, un nombre, un nom de registre, etc. Il existe diverses façons pour cela : chacune de ces façon va permettre d'interpréter le contenu de la partie variable comme étant une adresse, une constante, un nom de registre, etc, ce qui nous permettra de localiser la ou les donnée de notre instruction. Ces diverses manières d'interpréter notre partie variable pour en exploiter son contenu s'appellent des **modes d'adressage**. Pour résumer, ce mode d'adressage est une sorte de recette de cuisine capable de dire où se trouve la ou les données nécessaires pour exécuter une instruction. De plus, notre mode d'adressage peut aussi préciser où stocker le résultat de l'instruction.

Ces modes d'adressage dépendent fortement de l'instruction qu'on veut faire exécuter et du processeur. Certaines instructions supportent certains modes d'adressage et pas d'autres, voir mixent plusieurs modes d'adressages : les instructions manipulant plusieurs données peuvent parfois utiliser un mode d'adressage différent pour chaque donnée. Dans de tels cas, tout se passe comme si l'instruction avait plusieurs parties variables, nommées **opérandes**, contenant chacune soit une adresse, une donnée ou un registre. Pour comprendre un peu mieux ce qu'est un mode d'adressage, voyons quelques exemples de modes d'adressages assez communs et qui reviennent souvent.

Je vais donc parler des modes d'adressages suivants :

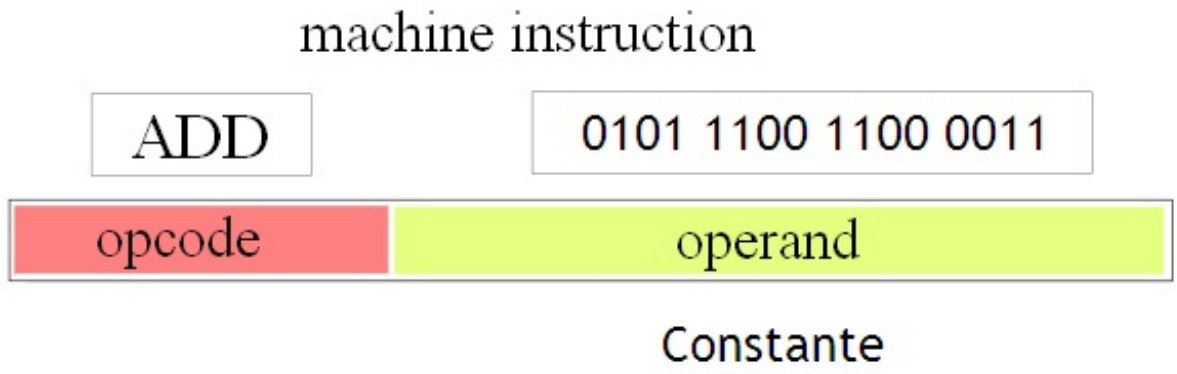
- implicite ;
- immédiat ;
- à registre ;
- absolus ;
- indirects à registres ;
- indirect avec auto-incrément ;
- indirect avec auto-décrément ;
- indexed absolute ;
- base + index ;
- base + offset ;
- base + index + offset.

Adressage implicite

Avec l'adressage implicite, la partie variable n'existe pas ! Il peut y avoir plusieurs raisons à cela. Il se peut que l'instruction n'aie pas besoin de données : une instruction de mise en veille de l'ordinateur, par exemple. Ensuite, certaines instructions n'ont pas besoin qu'on leur donne la localisation des données d'entrée et "savent" où est la ou les donnée(s). Comme exemple, on pourrait citer une instruction qui met tous les bits du registre d'état à zéro. Certaines instructions manipulant la pile sont adressées de cette manière : on connait d'avance l'adresse de la base ou du sommet de la pile. Pour rappel, celle-ci est stockée dans quelques registres du processeur.

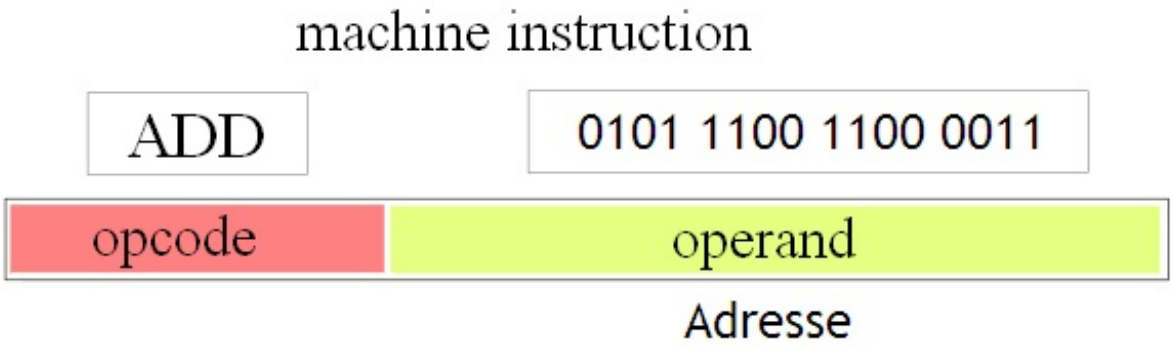
Adressage immédiat

Avec l'adressage immédiat, la partie variable est une constante. Celle-ci peut être un nombre, un caractère, un nombre flottant, etc. Avec ce mode d'adressage, notre donnée est chargée en même temps que l'instruction et est placée dans la partie variable.

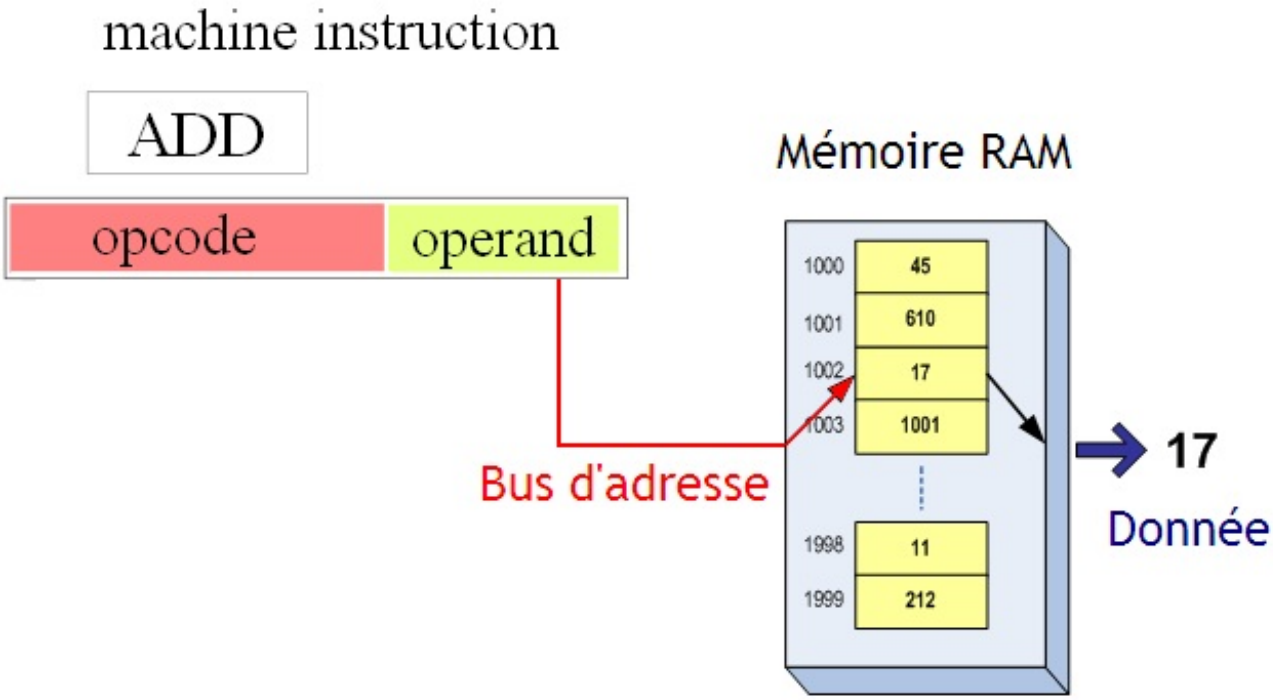


Adressage direct

Passons maintenant à l'adressage absolu, aussi appelé adressage direct. Avec lui, la partie variable est l'adresse de la donnée à laquelle accéder.



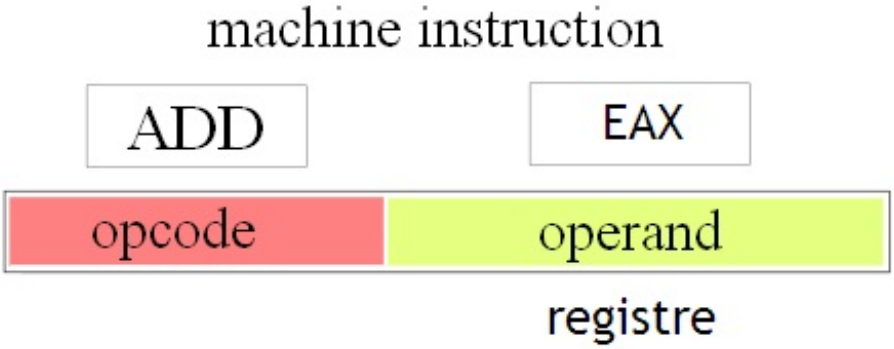
Cela permet parfois de lire une donnée directement depuis la mémoire sans devoir la copier dans un registre.



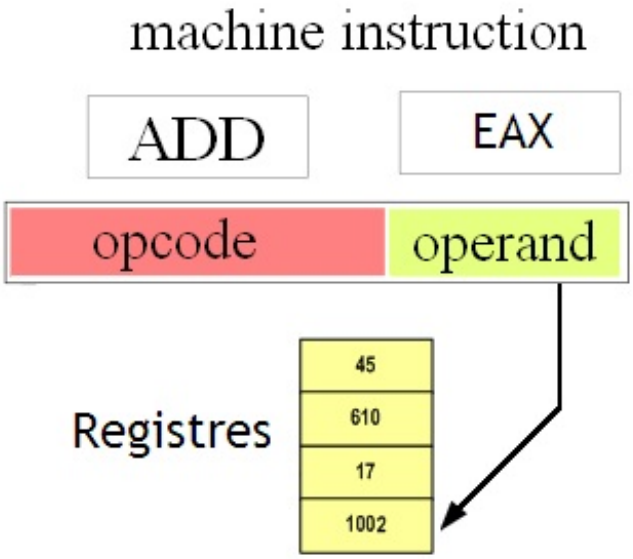
Ce mode d'adressage ne sert que pour les données dont l'adresse est fixée une bonne fois pour toute. Les seules données qui respectent cette condition sont les données placées dans la mémoire statique (souvenez-vous du chapitre précédent : on avait parlé des mémoires programme, statique, de la pile et du tas). Pour les programmeurs, cela correspond aux variables globales et aux variables statiques, ainsi qu'à certaines constantes (les chaines de caractères constantes, par exemple). Bien peu de données sont stockées dans cette mémoire statique, ce qui fait que ce mode d'adressage a tendance à devenir de plus en plus marginal.

Adressage inhérent

Avec le mode d'adressage inhérent, la partie variable va identifier un registre qui contient la donnée voulue.



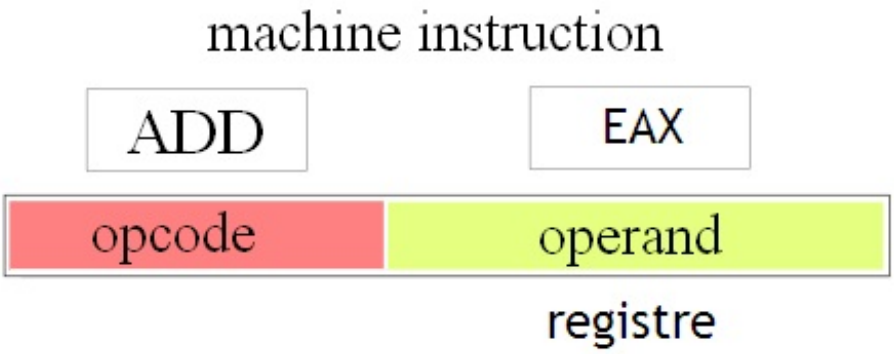
Mais identifier un registre peut se faire de différentes façons. On peut soit utiliser des noms de registres, ou encore identifier nos registres par des adresses mémoires. Le mode d'adressage inhérent n'utilise que des noms de registres.



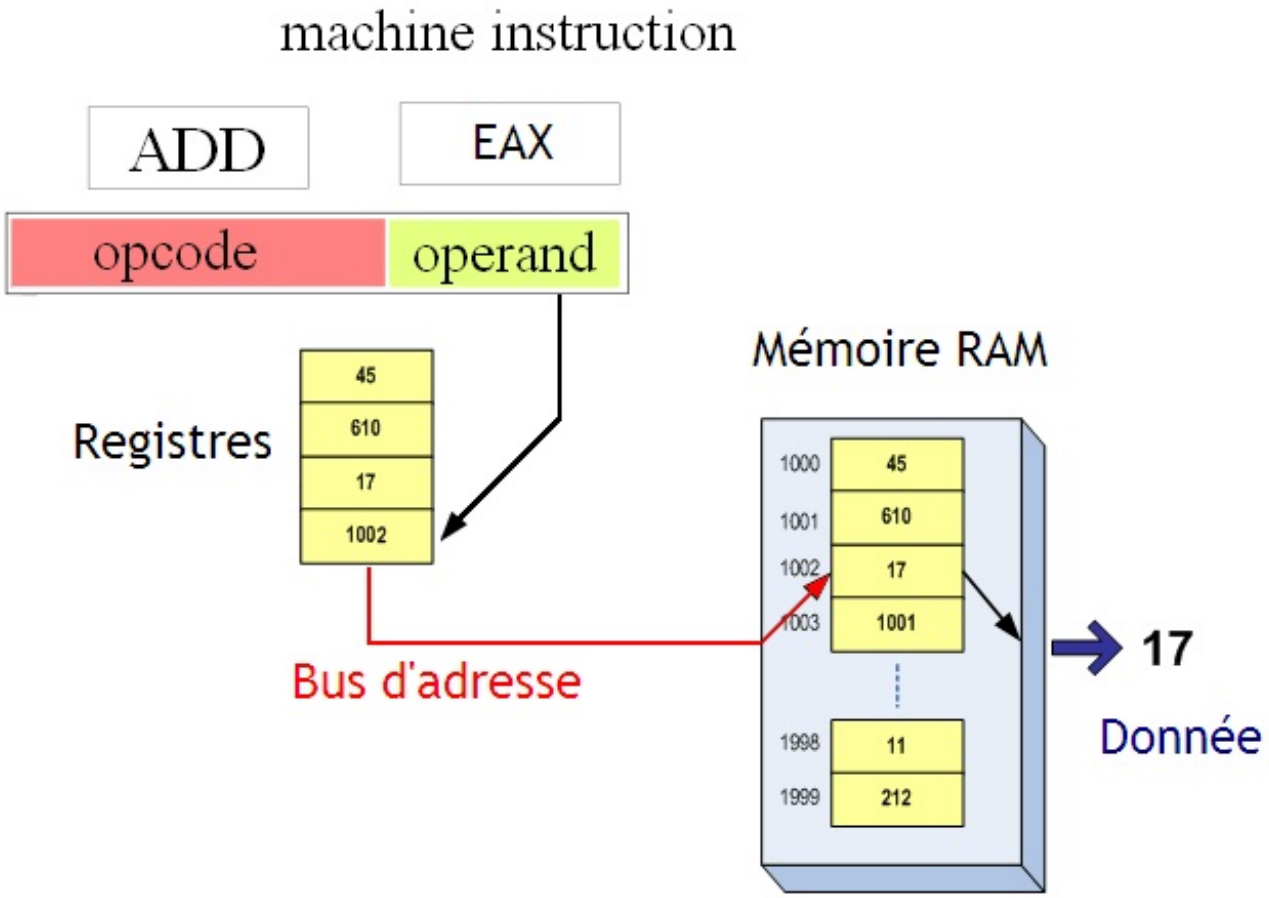
Adressage indirect à registre

Dans certains cas, les registres généraux du processeur peuvent stocker des adresses mémoire. Après tout, une adresse n'est rien d'autre qu'un nombre entier ayant une signification spéciale, et utiliser un registre censé stocker des nombres entiers pour stocker une adresse n'a rien de choquant. Ces adresses sont alors manipulables comme des données, et on peut leur faire subir quelques manipulations arithmétiques, comme des soustractions et des additions.

On peut alors décider à un moment ou un autre d'accéder au contenu de l'adresse stockée dans un registre : c'est le rôle du **mode d'adressage indirect à registre**. Ici, la partie variable permet d'identifier un registre contenant l'adresse de la donnée voulue.



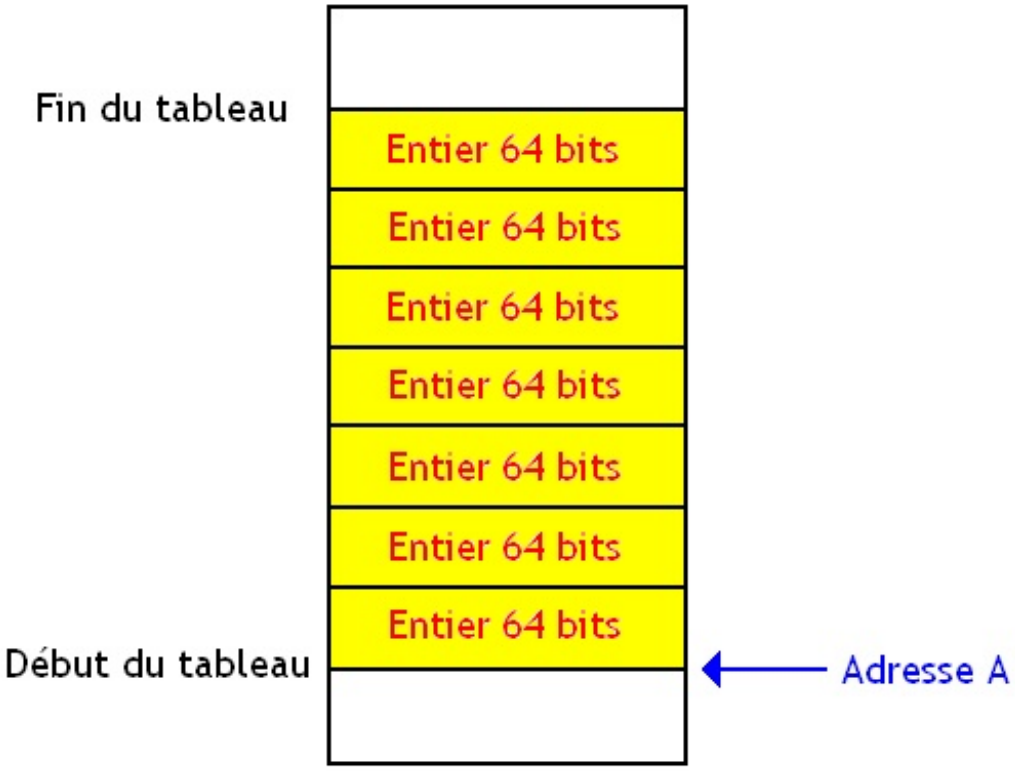
Si on regarde uniquement l'instruction telle qu'elle est en mémoire, on ne voit aucune différence avec le mode d'adressage inhérent vu juste au-dessus. La différence viendra de ce qu'on fait de ce nom de registre : le nom de registre n'est pas interprété de la même manière. Avec le mode d'adressage inhérent, le registre indiqué dans l'instruction contiendra la donnée à manipuler. Avec le mode d'adressage indirect à registre, la donnée sera placée en mémoire, et le registre contiendra l'adresse de la donnée.



Le mode d'adressage indirect à registre permet d'implémenter de façon simple ce qu'on appelle les **pointeurs**. Au début de l'informatique, les processeurs ne possédaient pas d'instructions ou de modes d'adressages pour gérer les pointeurs. On pouvait quand même gérer ceux-ci, en utilisant l'adressage direct. Mais dans certains cas, forçait l'utilisation de *self-modifying code*, c'est à dire que le programme devait contenir des instructions qui devaient modifier certaines instructions avant de les exécuter ! En clair, le programme devait se modifier tout seul pour faire ce qu'il faut. L'invention de ce mode d'adressage a permit de faciliter le tout : plus besoin de *self-modifying code*.

Pour donner un exemple, on peut citer l'exemple des **tableaux**. Un tableau est un ensemble de données de même taille rangées les unes à la suite des autres en mémoire.

- Première chose : **chaque donnée (on dit aussi élément) d'un tableau prend un nombre d'octets fixé une fois pour toute**. Généralement, une donnée prend entre 1, 2, 4, 8 octets : ce sont des nombres qui sont une puissance de deux.
- Deuxième chose, **ces données sont rangées les unes à coté des autres en mémoire** : on ne laisse pas le moindre vide. Les unes à coté des autre signifie dans des adresses mémoires consécutives.

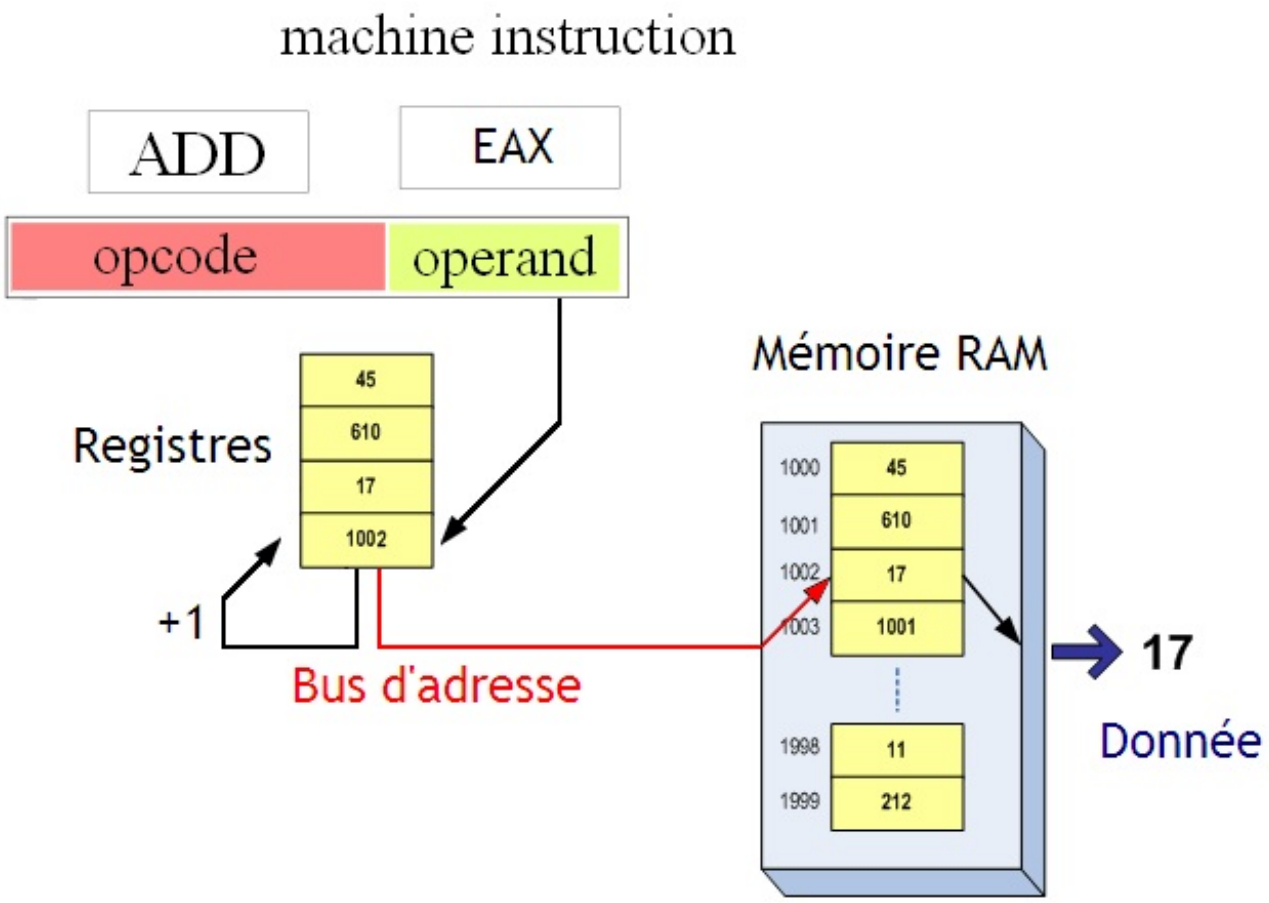


Exemple avec un tableau d'entiers prenant chacun 8 octets.

Stocker des données dans un tableau ne sert à rien si on ne peut pas les manipuler : le processeur doit connaitre l'adresse de l'élément qu'on veut lire ou écrire pour y accéder. Cette adresse peut se calculer assez simplement, en connaissant l'adresse du début du tableau, la longueur de chaque élément, ainsi que du numéro de l'élément dans notre tableau. Le seul problème, c'est qu'une fois calculée, notre adresse se retrouve dans un registre, et qu'il faut trouver un moyen pour y accéder. Et c'est là que le mode d'adressage indirect à registre intervient : une fois que l'adresse est calculée, elle est forcément stockée dans un registre : le mode d'adressage indirect à registre permet d'accéder à cette adresse directement. Sans ce mode d'adressage, on serait obligé d'utiliser une instruction utilisant le mode d'adressage direct, et de modifier l'adresse incorporée dans l'instruction avec du *self-modifying code*. Imaginez l'horreur.

Register Indirect Autoincrement/Autodecrement

Ce mode d'adressage existe aussi avec une variante : l'instruction peut automatiquement augmenter ou diminuer le contenu du registre d'une valeur fixe. Cela permet de passer directement à l'élément suivant ou précédent dans un tableau. Ce mode d'adressage a été inventé afin de faciliter le parcourt des tableaux. Il n'est pas rare qu'un programmeur aie besoin de traiter tous les éléments d'un tableau. Pour cela, il utilise une suite d'instructions qu'il répète sur tous les éléments : il commence par traiter le premier, passe au suivant, et continue ainsi de suite jusqu'au dernier. Ces modes d'adressage permettent d'accélérer ces parcourt en incrémentant ou décrémentant l'adresse lors de l'accès à notre élément.



Indexed Absolute

D'autres modes d'adressage permettent de faciliter la manipulations des tableaux. Ces modes d'adressage permettent de faciliter le calcul de l'adresse d'un élément du tableau. Reste à savoir comment ce calcul d'adresse est fait. Sachez que pour cela, chaque élément d'un tableau reçoit un nombre, un indice, qui détermine sa place dans le tableau : l'élément d'indice 0 est celui qui est placé au début du tableau, celui d'indice 1 est celui qui le suit immédiatement après dans la mémoire, etc. On doit donc calculer son adresse à partir de l'indice et d'autres informations. Pour cela, on utilise le fait que les éléments d'un tableau ont une taille fixe et sont rangés dans des adresses mémoires consécutives.

Prenons un exemple : un tableau d'entiers, prenant chacun 4 octets. Le premier élément d'indice zéro est placé à l'adresse A : c'est l'adresse à laquelle commence le tableau en mémoire. Le second élément est placé 4 octets après (vu que le premier élément prend 4 octets) : son adresse est donc $A + 4$. Le second élément est placé 4 octets après le premier élément, ce qui donne l'adresse $(A + 4) + 4$.

Si vous continuez ce petit jeu pour quelques valeurs, on obtiendrait quelque chose dans le genre :

Indice i	Adresse de l'élèment
0	A
1	A+4
2	A+8
3	A+12
4	A+16
5	A+20
...	...

Vous remarquerez surement quelque chose sur l'adresse de l'élément d'indice i, si vous vous souvenez que l'entier de notre

exemple fait 4 octets.

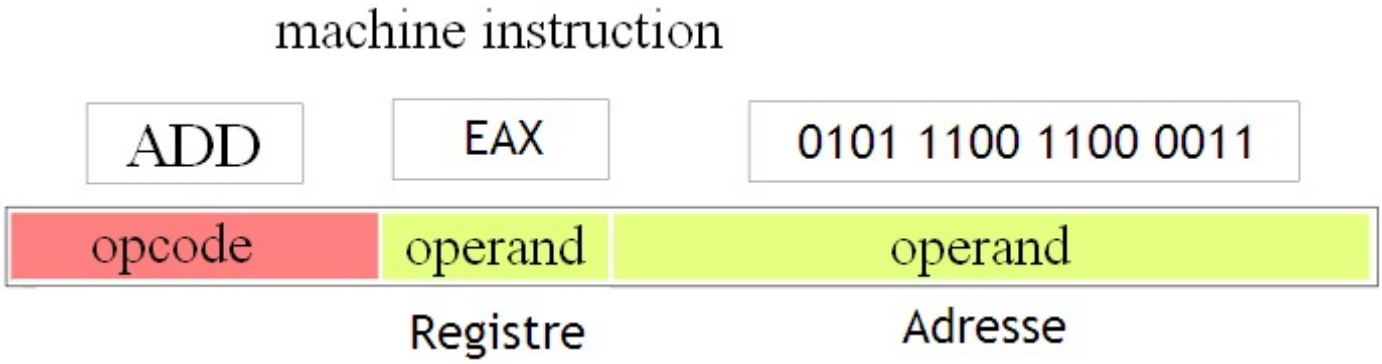
Indice i	Adresse de l'élèment
0	A + (0 * 4)
1	A + (1 * 4)
2	A + (2 * 4)
3	A + (3 * 4)
4	A + (4 * 4)
5	A + (5 * 4)
...	...

On peut formaliser cette remarque mathématiquement en posant L la longueur d'un élément du tableau, i l'indice de cet élément, et A l'adresse de début du tableau (l'adresse de l'élément d'indice zéro).

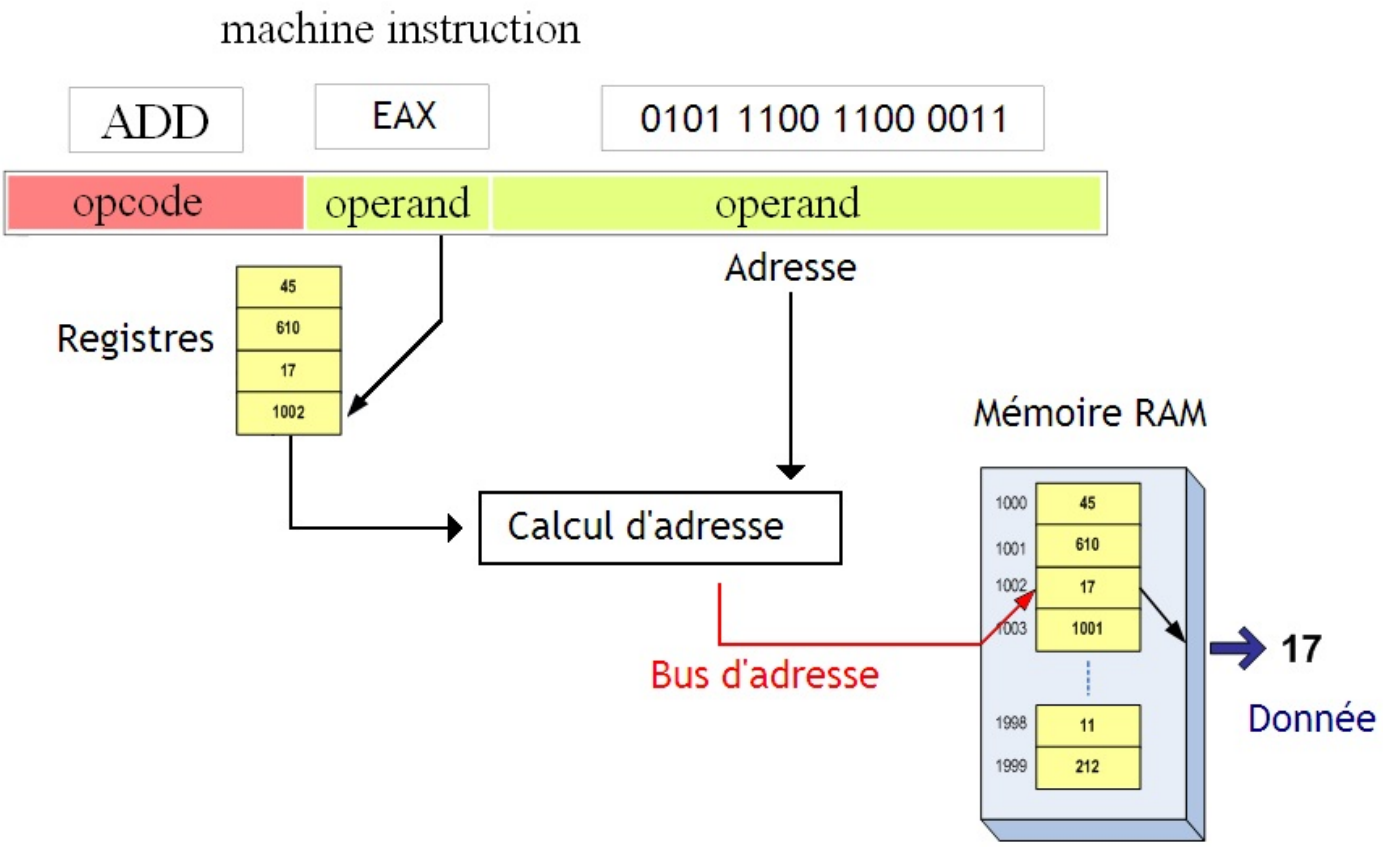
l'adresse de l'élément d'indice i vaut toujours $A + L \times i$.

Pour éviter d'avoir à calculer les adresses à la main avec le mode d'adressage *register indirect*, on a inventé un mode d'adressage pour combler ce manque : le mode d'adressage *Indexed Absolute*.

Celui-ci fournit l'adresse de base du tableau, et un registre qui contient l'indice.



A partir de ces deux données, l'adresse de l'élément du tableau est calculée, envoyée sur le bus d'adresse, et l'élément est récupéré.

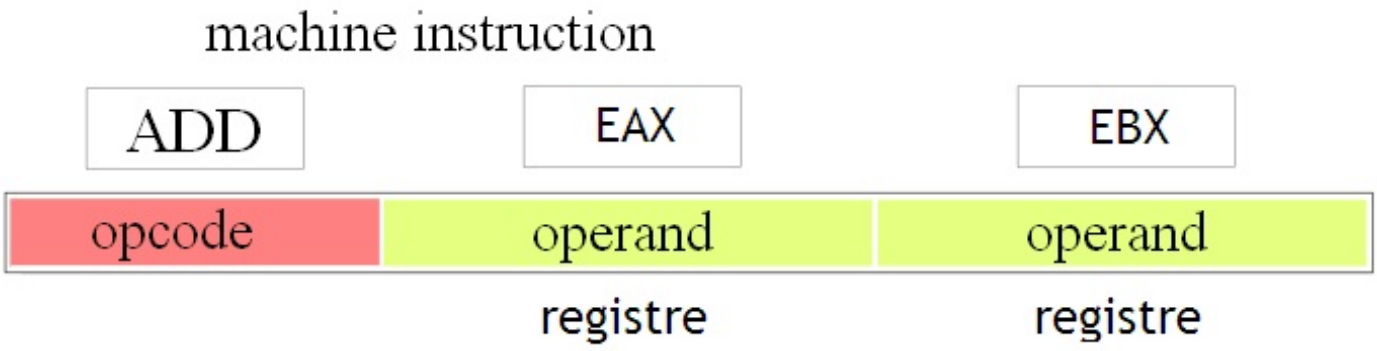


Base plus index

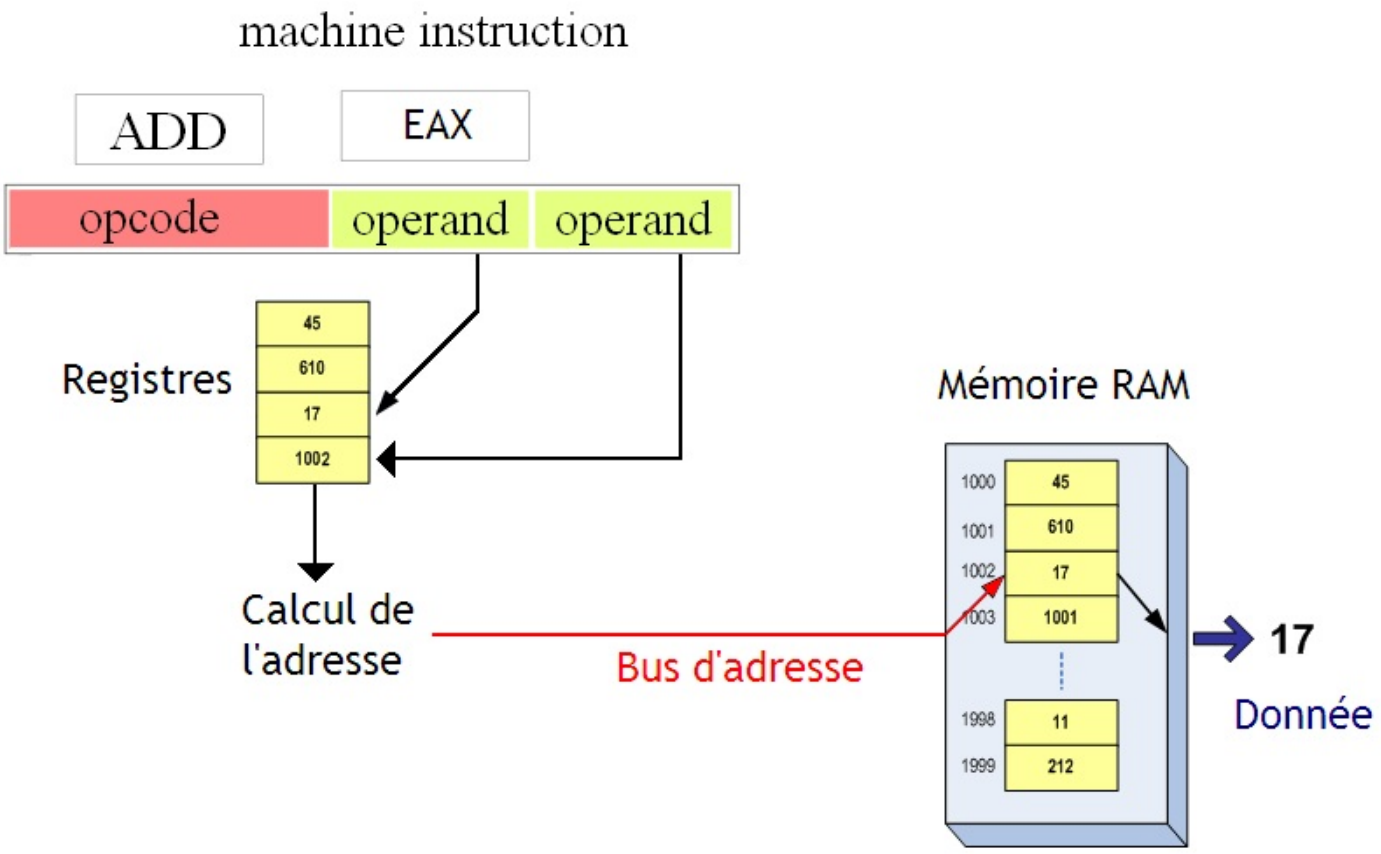
Le mode d'adressage *Indexed Absolute* vu plus haut ne marche que pour des tableaux dont l'adresse est fixée une bonne fois pour toute. Ces tableaux sont assez rare : ils correspondent aux tableaux de taille fixe, déclarée dans la mémoire statique (souvenez-vous du chapitre précédent). Et croyez moi, ces tableaux ne forment pas la majorité de l'espèce. La majorité des tableaux sont des tableaux dont l'adresse n'est pas connue lors de la création du programme : ils sont déclarés sur la pile ou dans le tas, et leur adresse varie à chaque exécution du programme. On peut certes régler ce problème en utilisant du *self-modifying code*, mais ce serait vendre son âme au diable !

Pour contourner les limitations du mode d'adressage *Indexed Absolute*, on a inventé le mode d'adressage *Base plus index*. Avec ce dernier, l'adresse du début du tableau n'est pas stockée dans l'instruction elle-même, mais dans un registre. Elle peut donc varier autant qu'on veut.

Ce mode d'adressage spécifie deux registres dans sa partie variable : un registre qui contient l'adresse de départ du tableau en mémoire : le **registre de base** ; et un qui contient l'indice : le **registre d'index**.



Le processeur calcule alors l'adresse de l'élément voulu à partir du contenu de ces deux registres, et accède à notre élément. En clair : notre instruction ne fait pas que calculer l'adresse de l'élément : elle va aussi le lire ou l'écrire.



Ce mode d'adressage possède une variante qui permet de vérifier qu'on ne "déborde pas" du tableau, en calculant par erreur une adresse en dehors du tableau, à cause d'un indice erroné, par exemple. Accéder à l'élément 25 d'un tableau de seulement 5 élément n'a pas de sens et est souvent signe d'une erreur. Pour cela, l'instruction peut prendre deux opérandes supplémentaires (qui peuvent être constantes ou placées dans deux registres). Si cette variante n'est pas supportée, on doit faire ces vérifications à la main. Parfois, certains processeurs implémentent des instructions capables de vérifier si les indices des tableaux sont corrects. Ces instructions sont capables de vérifier si un entier (l'indice) dépasse d'indice maximal autorisé, et qui effectuent un branchement automatique si l'indice n'est pas correct. L'instruction BOUND sur le jeu d'instruction x86 en est un exemple.

Base + Offset

Les tableaux ne sont pas les seuls regroupements de données utilisés par les programmeurs. Nos programmeurs utilisent souvent ce qu'on appelle des **structures**. Ces structures servent à créer des données plus complexe que celles que le processeur peut supporter. Comme je l'ai dit plus haut, notre processeur ne gère que des données simples : des entiers, des flottants ou des caractères. Pour créer des types de données plus complexe, on est obligé de regrouper des données de ce genre dans un seul bloc de mémoire : on crée ainsi une structure.

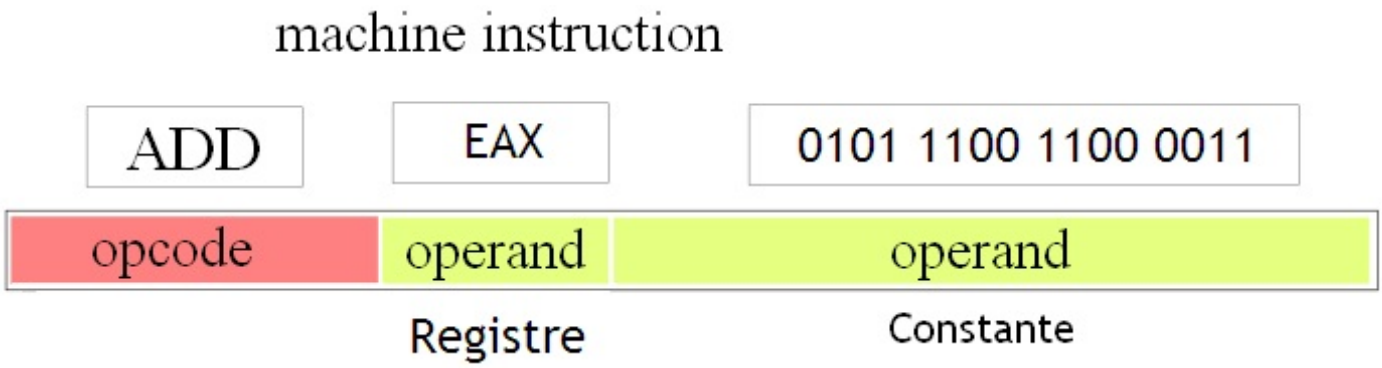
Par exemple, voici ce que donnerais une structure composée d'un entier, d'un flottant simple précision, et d'un caractère :

Octet 1	Octet 2	Octet 3	Octet 4	Octet 5	Octet 6	Octet 7	Octet 8	Octet 9
Adresse A	**Adresse A + 1**	**Adresse A + 2**	**Adresse A + 3**	**Adresse A + 4**	**Adresse A + 5**	**Adresse A + 6**	**Adresse A + 7**	**Adresse A + 8**
Entier 32 bits	Entier 32 bits	Entier 32 bits	Entier 32 bits	Flottant simple précision	Flottant simple précision	Flottant simple précision	Flottant simple précision	Caractère 8 bits

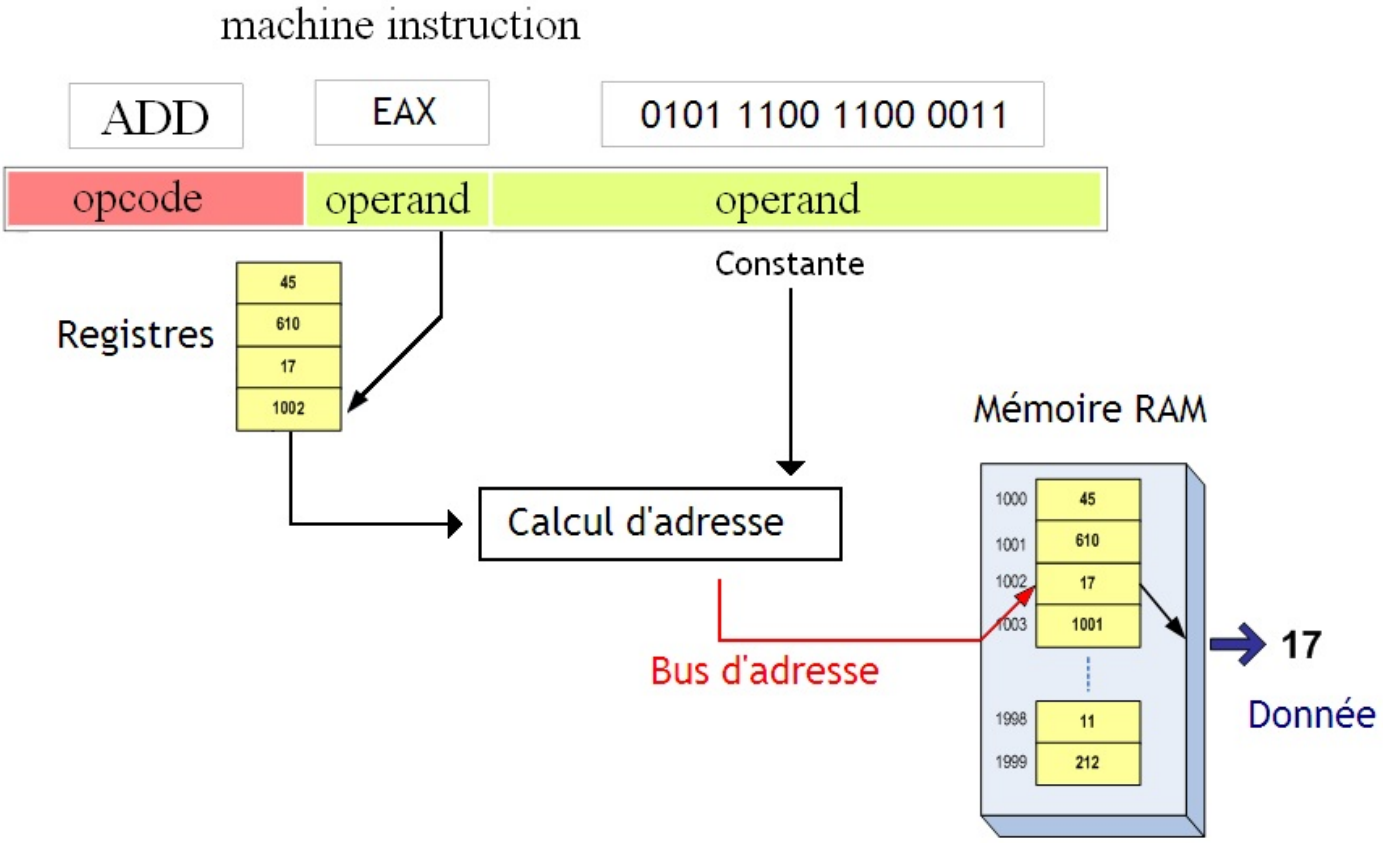
Mais le processeur ne peut pas manipuler ces structures : il est obligé de manipuler les données élémentaires qui la constituent unes par unes. Pour cela, il doit calculer leur adresse. Ce qui n'est pas très compliqué : une donnée a une place prédéterminée dans une structure. Elle est donc a une distance fixe du début de la structure.

Calculer l'adresse d'un élément de notre structure se fait donc en ajoutant une constante à l'adresse de départ de la structure. Et

c'est ce que fait le mode d'adressage *Base + Offset*. Celui-ci spécifie un registre qui contient l'adresse du début de la structure, et une constante.



Ce mode d'adressage va non seulement effectuer ce calcul, mais il va aussi aller lire (ou écrire) la donnée adressée.



Base + Index + offset

Certains processeurs vont encore plus loin : ils sont capables de gérer des tableaux de structures ! Ce genre de prouesse est possible grâce au mode d'adressage *Base + Index + offset*. Avec ce mode d'adressage, on peut calculer l'adresse d'une donnée placée dans un tableau de structure assez simplement : on calcule d'abord l'adresse du début de la structure avec le mode d'adressage Base + Index, et ensuite on ajoute une constante pour repérer la donnée dans la structure. Et le tout, en un seul mode d'adressage. Autant vous dire que ce mode d'adressage est particulièrement complexe, et qu'on n'en parlera pas plus que cela.

Autres

D'autres modes d'adressages existent, et en faire une liste exhaustive serait assez long. Ce serait de plus inutile, vu que la plupart sont de toute façon obsolètes. Des modes d'adressage comme le *Memory indirect* ne servent plus à grand chose de nos jours.

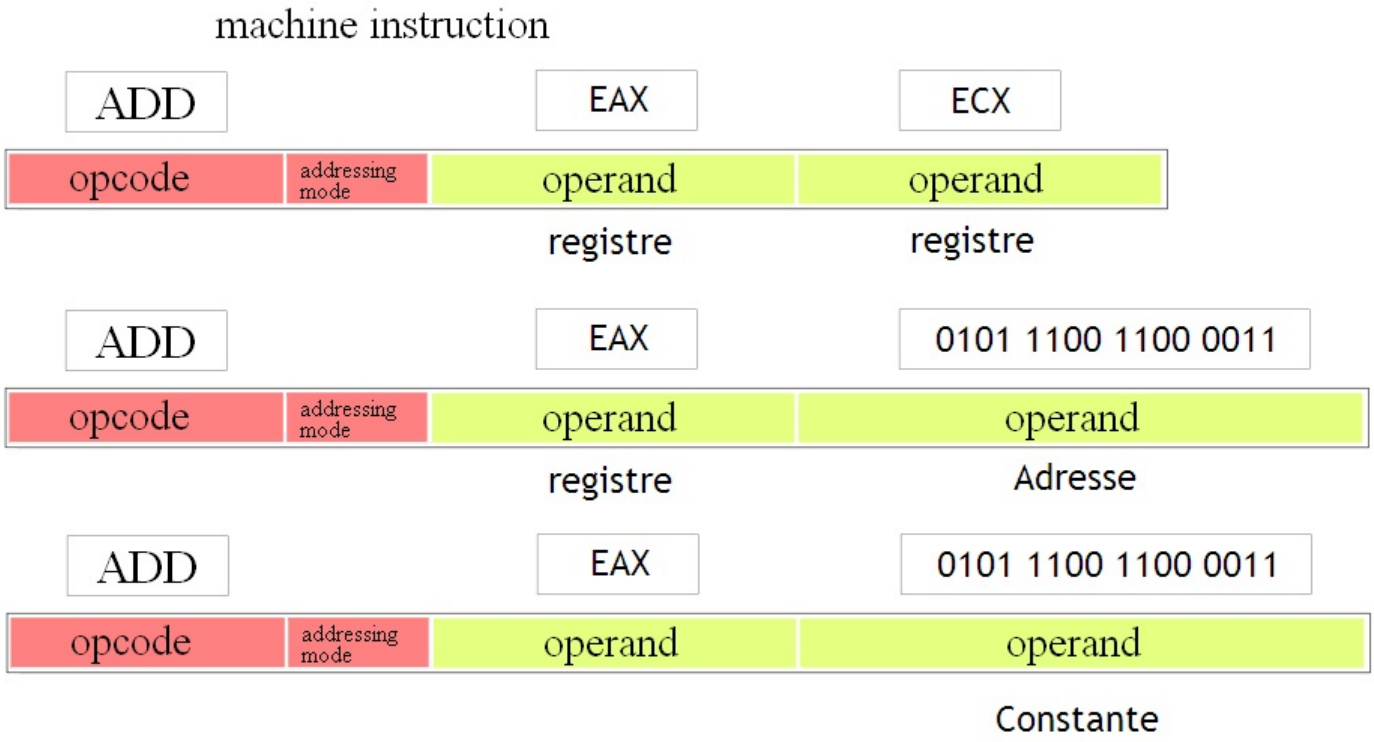
Encodage du mode d'adressage

Dans le paragraphe du dessus, on a vu les divers modes d'adressages les plus utilisés. Mais nous n'avons pas encore parlé de quelque chose de fondamental : comment préciser quel mode d'adressage notre instruction utilise ? Et bien sachez que cela se fait de diverses manières suivant les instructions.

Explicite

Nous allons voir un premier cas : celui des instructions pouvant gérer plusieurs modes d'adressages par opérandes. Prenons un exemple : je dispose d'une instruction d'addition. Les deux opérandes de mon instruction peuvent être soit des registres, soit un registre et une adresse, soit un registre et une constante. La donnée à utiliser sera alors chargée depuis la mémoire ou depuis un registre, ou prise directement dans l'instruction, suivant l'opérande utilisée. Dans un cas pareil, je suis obligé de préciser quel est le mode d'adressage utiliser. Sans cela, je n'ai aucun moyen de savoir si la seconde opérande est un registre, une constante, ou une adresse. Autant je peux le savoir pour la première opérande : c'est un registre, autant le mode d'adressage de la seconde m'est inconnu.

On est dans un cas dans lequel certaines opérandes ont plusieurs modes d'adressage. Pour ces instructions, le mode d'adressage doit être précisé dans notre instruction. Quelques bits de l'instruction doivent servir à préciser le mode d'adressage. Ces bits peuvent être placés dans l'opcode, ou dans quelques bits à part, séparés de l'opcode et des opérandes (généralement, ces bits sont intercalés entre l'opcode et les opérandes).

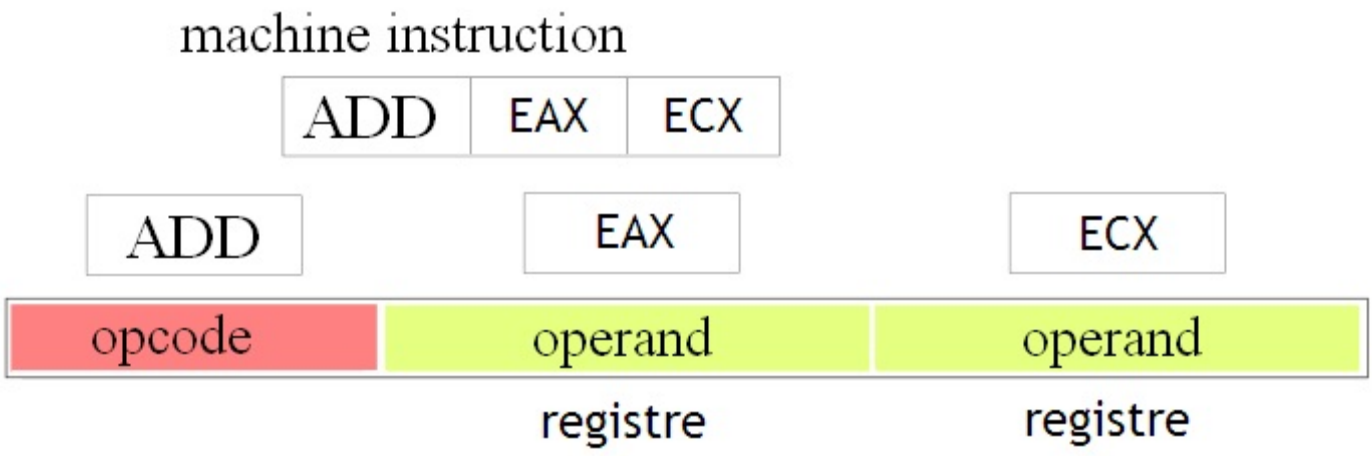


Implicite

Dans le second cas, notre instruction ne peut gérer qu'un seul mode d'adressage par opérande, toujours le même. Prenons un exemple : j'ai un processeur RISC dont toutes les instructions arithmétiques ne peuvent manipuler que des registres. Pas de mode d'adressage immédiat, ni absolu ni quoique ce soit : les opérandes des instructions arithmétiques utilisent toutes le mode d'adressage à registre.

Prenons un autre exemple : l'instruction `Load`. Cette instruction va lire le contenu d'une adresse mémoire et stocker celui-ci dans un registre. Cette instruction a deux opérandes prédéfinies : un registre, et une adresse mémoire. Notre instruction utilise donc le mode d'adressage absolu pour la source de la donnée à lire, et un nom de registre pour la destination du résultat. Et cela ne change jamais : notre instruction a ses modes d'adressages prédéfinis, sans aucune possibilité de changement.

Dans un cas pareil, si chaque opérande a un mode d'adressage prédéterminé, pas besoin de le préciser vu que celui-ci ne change jamais. Celui-ci peut être déduit automatiquement en connaissant l'instruction : il est plus ou moins implicite. On n'a pas besoin d'utiliser des bits de notre instruction pour préciser le mode d'adressage, qui peut être déduit à partir de l'Opcode.



Jeux d'instructions et modes d'adressages

Le nombre de mode d'adressages différents gérés par un processeur dépend fortement de son jeu d'instruction. Les processeurs CISC ont souvent beaucoup de modes d'adressages. C'est tout le contraire des processeurs RISC, qui ont très peu de modes d'adressage : cela permet de simplifier la conception du processeur au maximum. Et cela rend difficile la programmation en assembleur : certains modes d'adressages facilitent vraiment la vie (le mode d'adressage indirect à registre, notamment).

Sur certains processeurs, chaque instruction définie dans le jeu d'instruction peut utiliser tous les modes d'adressages supportés par le processeur : on dit que le jeu d'instruction du processeur est **orthogonal**. Les jeu d'instructions orthogonaux sont une caractéristique des processeurs CISC, et sont très rares chez les processeurs RISC.

Longueur d'une instruction

Une instruction va prendre un certain nombre de bits en mémoire. On dit aussi qu'elle a une certaine longueur. Et cette longueur dépend de l'instruction et de ses opérandes. Les opérandes d'une instruction n'ont pas la même taille. Ce qui fait que nos instructions auront des tailles différentes, si elles utilisent des opérandes différentes. Par exemple, une opérande contenant une adresse mémoire (adressage direct) prendra plus de place qu'une opérande spécifiant un registre : pour un processeur de 64 registres, il suffira d'encoder de quoi spécifier 64 registres. Par contre, une adresse permet souvent de préciser bien plus que 64 cases mémoires et prend donc plus de place. Généralement, l'adressage par registre et l'adressage indirect à registre permettent d'avoir des opérandes petites comparé aux modes d'adressage direct et immédiat. Mais le mode d'adressage implicite est celui qui permet de se passer complètement de partie variable et est donc le plus économe en mémoire.

Quoiqu'il en soit, on pourrait croire que la taille d'une instruction est égale à celle de ses opérandes + celle de son opcode. Mais c'est faux. C'est vrai sur certains processeurs, mais pas sur tous. Certains processeurs ont des instructions de taille fixe, peut importe la taille de leurs opérandes. D'autres processeurs utilisent des instructions de taille variable, pour éviter de gaspiller et prendre juste ce qu'il faut de mémoire pour stocker l'opcode et les opérandes.

Longueur variable

Sur certains processeurs, cette longueur est variable : toutes les instructions n'ont pas la même taille. Ainsi, une instruction d'addition prendra moins de bits qu'une instruction de branchement, par exemple. Cela permet de gagner un peu de mémoire : avoir des instructions qui font entre 2 et 3 octets est plus avantageux que de tout mettre sur 3 octets. En contrepartie, calculer l'adresse de la prochaine instruction est assez compliqué : la mise à jour du *Program Counter* nécéssite pas mal de travail.

Les processeurs qui utilisent ces instructions de longueur variable sont souvent des processeurs CISC. Il faut dire que les processeurs CISC ont beaucoup d'instructions, ce qui fait que l'opcode de chaque instruction est assez long et prend de la mémoire. Avoir des instructions de longueur variable permet de limiter fortement la casse, voir même d'inverser la tendance. La taille de l'instruction dépend aussi du mode d'adressage : la taille d'une opérande varie suivant sa nature (une adresse, une constante, quelques bits pour spécifier un registre, voire rien).

Longueur fixe

Sur d'autres processeurs, cette longueur est généralement fixe. 2videmment, cela gâche un peu de mémoire comparé à des instructions de longueur variable. Mais cela permet au processeur de calculer plus facilement l'adresse de l'instruction suivante en faisant une simple addition. Et cela a d'autres avantages, dont on ne parlera pas ici.

Les instructions de longueur fixe sont surtout utilisées sur les processeurs RISC. Sur les processeurs RISC, l'opcode prend peu

de place : il y a peu d'instructions différentes à coder, donc l'opcode est plus court, et donc on préfère simplifier le tout et utiliser des instructions de taille fixe.

Classes d'architectures

Tous ces modes d'adressage ne sont pas supportés par tous les processeurs ! En fait, il existe plusieurs types d'architectures, définies par leurs modes d'adressages possibles. Certaines ne supportent pas certains modes d'adressage. Et pour s'y repérer, on a décider de classifier un peu tout ça.

Il existe donc 5 classes d'architectures :

- à accès mémoires strict ;
- à accumulateur ;
- a pile ;
- Load-Store ;
- registres-mémoire.

A accès mémoire strict

Dans cette architecture ci, il n'y a pas de registres généraux : les instructions n'accèdent qu'à la mémoire principale. Néanmoins, les registres d'instruction et pointeur d'instruction existent toujours. Les seules opérandes possibles pour ces processeurs sont des adresses mémoire, ce qui fait qu'un mode d'adressage est très utilisé : l'adressage absolu. Ce genre d'architectures est aujourd'hui tombé en désuétude depuis que la mémoire est devenue très lente comparé au processeur.

A pile

Dans les architectures à pile, il n'y a pas de registres stockant de données : les instructions n'accèdent qu'à la mémoire principale, exactement comme pour les architectures à accès mémoire strict. Néanmoins, ces machines fonctionnent différemment. Ces processeurs implémentent une pile, et écrivent donc tous leurs résultats en mémoire RAM. Et oui, vous ne vous êtes pas trompés : il s'agit bien de nos bonnes vielles machines à pile, vues il y a quelques chapitres.

Push et Pop

Ces architectures ont besoin d'instructions pour transférer des données entre la pile et le reste de la mémoire. Pour cela, ces processeurs disposent d'instructions spécialisées pour pouvoir empiler une donnée au sommet de la pile : `push` ; et une instruction pour dépiler la donnée au sommet de la pile et la sauvegarder en mémoire : `pop`.

Les instructions `push` et `pop` vont aller lire ou écrire à une adresse mémoire bien précise. Cette adresse spécifie l'adresse de la donnée à charger pour `push` et l'adresse à laquelle sauvegarder le sommet de la pile pour `pop`. Cette adresse peut être précisée via différents modes d'adressages : absolus, Base + Index, etc. L'instruction `push` peut éventuellement empiler une constante, et utilise dans ce cas le mode d'adressage immédiat.

Instructions arithmétiques

Toutes les instructions arithmétiques et logiques vont aller chercher leurs opérandes sur le sommet de la pile. Ces instructions vont donc dépiler un certain nombre d'opérandes (1 ,2 voire 3), et vont stocker le résultat au sommet de la pile. Le sommet de la pile est adressé de façon implicite : le sommet de la pile est toujours connu (son adresse est stockée dans un registre dédié et on n'a donc pas besoin de la préciser par un mode d'adressage).

Notre instructions arithmétiques et logiques se contentent d'un opcode, vu que les opérandes sont adressées implicitement. C'est pour cela que sur ces processeurs, la mémoire utilisée par le programme est très faible.

Sur un ordinateur qui n'est pas basé sur une architecture à pile, on aurait dû préciser la localisation des données en ajoutant une partie variable à l'opcode de l'instruction, augmentant ainsi la quantité de mémoire utilisée pour stocker celle-ci.

Machines à pile 1 et 2 adresses

Dans ce que je viens de dire au-dessus, les machine à pile que je viens de décrire ne pouvait pas manipuler autre chose que des données placées sur la pile : ces machines à pile sont ce qu'on appelle des machines zéro adresse. Toutefois, certaines machines à pile autorisent certaines instructions à pouvoir, si besoin est, préciser l'adresse mémoire d'une (voire plusieurs dans certains cas) de leurs opérandes. Ainsi, leurs instructions peuvent soit manipuler des données placées sur la pile, soit une donnée placée sur la pile et une donnée qui n'est pas sur la pile, mais dans la mémoire RAM. Ces machines sont appelées des machines à pile une adresse.

A accumulateur unique

Sur certains processeurs, les résultats d'une opération ne peuvent être enregistrés que dans un seul registre, prédéfini à l'avance : **l'accumulateur**. Cela ne signifie pas qu'il n'existe qu'un seul registre dans le processeur. Mais vu qu'une instruction ne peut pas modifier leur contenu, le seul moyen d'écrire dans ces registres est de lire une donnée depuis la mémoire et de stocker le résultat de la lecture dedans. Toute instruction va obligatoirement lire une donnée depuis cette accumulateur, et y écrire son résultat. Si l'instruction a besoin de plusieurs opérandes, elle va en stocker une dans cet accumulateur et aller chercher les autres dans la mémoire ou dans les autres registres.

Dans tous les cas, l'accumulateur est localisé grâce au mode d'adressage implicite. De plus, le résultat des instructions arithmétiques et logiques est stocké dans l'accumulateur, et on n'a pas besoin de préciser où stocker le résultat : pas de mode d'adressage pour le résultat.

Architectures 1-adresse

Historiquement, les premières architectures à accumulateur ne contenaient aucun autre registre : l'accumulateur était seul au monde. Pour faire ses calculs, notre processeur devait stocker une opérande dans l'accumulateur, et aller chercher les autres en mémoire. En conséquence, le processeur ne pouvait pas gérer certains modes d'adressages, comme le mode d'adressage à registre. Sur ces processeurs, les modes d'adressages supportés étaient les modes d'adressages implicite, absolus, et immédiat.

Ces architectures sont parfois appelées **architectures 1-adresse**. Cela vient du fait que la grosse majorité des instructions n'ont besoin que d'une opérande. Il faut dire que la majorité des instructions d'un processeur n'a besoin que de deux opérandes et ne fournissent qu'un résultat : pensez aux instructions d'addition, de multiplication, de division, etc. Pour ces opérations, le résultat ainsi qu'une des opérandes sont stockés dans l'accumulateur, et adressés de façon implicite. Il y a juste à préciser la seconde opérande à l'instruction, ce qui prend en tout une opérande.

Architectures à registres d'Index

Évidemment, avec ces seuls modes d'adressages, l'utilisation de tableaux ou de structures devenait un véritable calvaire. Pour améliorer la situation, ces processeurs à accumulateurs ont alors incorporés des registres d'*Index*, capables de stocker des indices de tableaux, ou des constantes permettant de localiser une donnée dans une structure. Ces registres permettaient de faciliter les calculs d'adresses mémoire.

Au départ, nos processeurs n'utilisaient qu'un seul registre d'*Index*, accessible et modifiable via des instructions spécialisées. Ce registre d'*Index* se comportait comme un second accumulateur, spécialisés dans les calculs d'adresses mémoire. Les modes d'adressages autorisés restaient les mêmes qu'avec une architecture à accumulateur normale. La seule différence, c'est que le processeur contenait de nouvelles instruction capables de lire ou d'écrire une donnée dans/depuis l'accumulateur, qui utilisaient ce registre d'*Index* de façon implicite.

Mais avec le temps, nos processeurs finirent par incorporer plusieurs de ces registres. Nos instructions de lecture ou d'écriture devaient alors préciser quel registre d'*Index* utiliser. Le mode d'adressage *Indexed Absolute* vit le jour. Les autres modes d'adressages, comme le mode d'adressage *Base + Index* ou indirects à registres étaient plutôt rares à l'époque et étaient difficiles à mettre en œuvre sur ce genre de machines.

Architectures 2,3-adresse

Ensuite, ces architectures s'améliorèrent un petit peu : on leur ajouta des registres capables de stocker des données. L'accumulateur n'était plus seul au monde. Mais attention : ces registres ne peuvent servir que d'opérande dans une instruction, et le résultat d'une instruction ira obligatoirement dans l'accumulateur. Ces architectures supportaient donc le mode d'adressage à registre.

Architectures registre-mémoire

C'est la même chose que l'architecture à accumulateur, mais cette fois, le processeur peut aussi contenir plusieurs autres registres généraux qui peuvent servir à stocker pleins de données diverses et variées. Le processeur peut donc utiliser plusieurs registres pour stocker des informations (généralement des résultats de calcul intermédiaires), au lieu d'aller charger ou stocker ces données dans la mémoire principale.

Ces architectures à registres généraux (ainsi que les architectures Load-store qu'on verra juste après) sont elles-même divisées en deux sous-classes bien distinctes : les architectures 2 adresses et les architectures 3 adresses. Cette distinction entre architecture 2 et 3 adresses permet de distinguer les modes d'adressages des opérations arithmétiques manipulant deux données : additions, multiplications, soustraction, division, etc. Ces instructions ont donc besoin d'adresser deux données différentes, et de stocker le résultat quelque part. Il leur faut donc préciser trois opérandes dans le résultat : la localisation des deux données à manipuler, et l'endroit où ranger le résultat.

Architectures 2 adresse

Sur les architectures deux adresses, l'instruction possède seulement deux opérandes pour les données à manipuler, l'endroit où ranger le résultat étant adressé implicitement. Plus précisément, le résultat sera stocké au même endroit que la première donnée qu'on manipule : cette donnée sera remplacée par le résultat de l'instruction.

Mnémonique/Opccode	Opérande 1	Opérande 2
DIV (Division)	Dividende / Résultat	Diviseur

Avec cette organisation, les instructions sont plus courtes. Mais elle est moins souple, vu que l'une des données utilisée est écrasée : si on a encore besoin de cette donnée après l'exécution de notre instruction, on est obligé de copier cette donnée dans un autre registre et faire travailler notre instruction sur une copie.

Architectures 3 adresse

Sur les architectures trois adresses, l'instruction possède trois opérandes : deux pour les données à manipuler, et une pour le résultat.

Mnémonique/Opccode	Opérande 1	Opérande 2	Opérande 3
DIV	Dividende	Diviseur	Résultat

Les instructions de ce genre sont assez longues, mais on peut préciser à quel endroit ranger le résultat. On n'est ainsi pas obligé d'écraser une des deux données manipulées dans certains cas, et stocker le résultat de l'instruction dans un registre inutilisé, préférer écraser une autre donnée qui ne sera pas réutilisée, etc. Ce genre d'architecture permet une meilleure utilisation des registres, ce qui est souvent un avantage. Mais par contre, les instructions deviennent très longues, ce qui peut devenir un vrai problème. Sans compter que devoir gérer trois modes d'adressages (un par opérande) au lieu de deux risque d'être assez couteux en circuits et en transistors : un circuit aussi complexe sera plus lent et coutera cher. Et ces désavantages sont souvent assez ennuyeux.

Load-store

Cette fois, la différence n'est pas au niveau du nombre de registres. Dans cette architecture, toutes les instructions arithmétiques et logiques ne peuvent aller chercher leurs données que dans des registres du processeurs.

Accès mémoires

Seules les instructions `load` et `store` peuvent accéder à la mémoire. `load` permet de copier le contenu d'une (ou plusieurs) adresse mémoire dans un registre, tandis que `store` copie en mémoire le contenu d'un registre. `load` et `store` sont des instructions qui prennent comme opérande le nom d'un registre et une adresse mémoire. Ces instructions peuvent aussi utiliser l'adressage indirect à registre ou tout autre mode d'adressage qui fournit une adresse mémoire.

Instructions arithmétiques et logiques

Toutes les autres instructions n'accèdent pas directement à la mémoire. En conséquence, ces instructions ne peuvent prendre que des noms de registres ou des constantes comme opérandes. Cela autorise les modes d'adressage immédiat et à registre. Il faut noter aussi que les architectures Load-store sont elles aussi classées en architectures à 2 ou 3 adresses. Tous les processeurs RISC inventés à ce jour sont basés sur une architecture Load-store.

Un peu de programmation !

Notre processeur va exécuter des programmes, fabriqués à l'aide de ce qu'on appelle un **langage de programmation**. Ces langages de programmations sont des langages qui permettent à un humain de programmer plus ou moins facilement un ordinateur pour que celui-ci fasse ce qu'on veut. Ces langages de programmations ont influencés de façon notable les jeux d'instructions des processeurs modernes : de nombreux processeurs implémentent des instructions spécialement conçues pour faciliter la traduction des "briques de base" de certains langages de programmation en une suite d'instructions machines. Ces fameuses "briques de base" sont ce que l'on appelle les **structures de contrôle**.

Autant prévenir tout de suite : j'ai fait en sorte que même quelqu'un qui ne sait pas programmer puisse comprendre cette partie. Ceux qui savent déjà programmer auront quand même intérêt à lire ce chapitre : il leur permettra de savoir ce qui se passe quand leur ordinateur exécute du code. De plus, ce chapitre expliquera beaucoup de notions concernant les branchements et les instructions de test, survolées au chapitre précédent, qui serviront plus tard quand on abordera la prédiction de branchement et d'autres trucs du même acabit. Et c'est sans compter que vous allez apprendre des choses intéressantes, comme l'utilité de la pile dans les architectures actuelles.

C'est un ordre, éxecution !

On va commencer ce chapitre par quelques rappels. Vous savez déjà qu'un programme est une suite d'instructions stockée dans la mémoire programme. Lorsqu'on allume le processeur, celui-ci charge automatiquement la première instruction du programme : il est conçu pour. Puis, il va passer à l'instruction suivante et l'exécuter. Notre processeur poursuivra ainsi de suite, en passant automatiquement à l'instruction suivante, et exécutera les instructions du programme les unes après les autres.

Il fera ainsi jusqu'à la dernière instruction de notre programme. Celle-ci est souvent une instruction qui permet de stopper l'exécution du programme au point où il en est (du moins si le programme ne boucle pas indéfiniment). Cette fameuse instruction d'arrêt est souvent exécutée par le programme, histoire de dire : "j'ai fini" ! Ou alors pour dire : "j'ai planté !" .

Certains processeurs modernes fonctionnent d'une manière légèrement différente de ce qu'on vient de voir. Par exemple, ils peuvent éxecuter plusieurs instructions à la fois, ou peuvent exécuter les instructions d'un programme dans un ordre différent de celui imposé par notre programme. On verra cela plus tard, dans la suite du tutoriel. Qui plus est, il existe une classe de processeurs un peu spéciaux, qui n'utilise pas de registre d'adresse d'instruction et n'exécute pas vraiment les instructions dans l'ordre imposé par un programme qui demanderait de les exécuter unes par unes, en série : il s'agit des fameuses architectures *dataflow* !

Program Counter

Il est évident que pour éxecuter une suite d'instructions dans le bon ordre, notre ordinateur doit savoir quelle est la prochaine instruction à exécuter. Il faut donc que notre processeur se souvienne de cette information quelque part : notre processeur doit donc contenir une mémoire qui stocke cette information. C'est le rôle du registre d'adresse d'instruction, aussi appelé ***Program Counter***. Ce registre stocke l'adresse de la prochaine instruction à exécuter. Cette adresse permet de localiser l'instruction suivante en mémoire. Cette adresse ne sort pas de nulle part : on peut la déduire de l'adresse de l'instruction en cours d'exécution par divers moyens plus ou moins simples qu'on verra dans la suite de ce tutoriel.

Ce calcul peut être fait assez simplement. Généralement, on profite du fait que ces instructions sont exécutées dans un ordre bien précis, les unes après les autres. Sur la grosse majorité des ordinateur, celles-ci sont placées les unes à la suite des autres dans l'ordre où elles doivent être exécutées. L'ordre en question est décidé par le programmeur. Un programme informatique n'est donc qu'une vulgaire suite d'instructions stockée quelque part dans la mémoire de notre ordinateur.

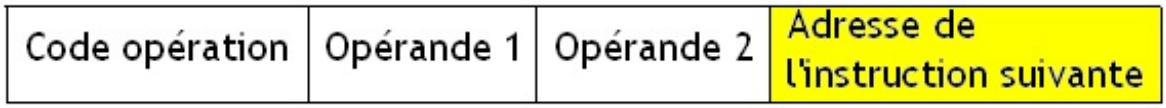
Par exemple :

Adresse	Instruction
0	Charger le contenu de l'adresse 0F05
1	Charger le contenu de l'adresse 0555
2	Additionner ces deux nombres
3	Charger le contenu de l'adresse 0555
4	Faire en XOR avec le résultat antérieur
...	...
5464	Instruction d'arrêt

En faisant ainsi, on peut calculer facilement l'adresse de la prochaine instruction en ajoutant la longueur de l'instruction juste chargée (le nombre de case mémoire qu'elle occupe) au contenu du registre d'adresse d'instruction. Dans ce cas, l'adresse de la prochaine instruction est calculée par un petit circuit combinatoire couplé à notre registre d'adresse d'instruction, qu'on appelle le compteur ordinal.

Les exceptions

Mais certains processeurs n'utilisent pas cette méthode. Mais il s'agit de processeurs particulièrement rares. Sur de tels processeurs, chaque instruction va devoir préciser où est la prochaine instruction. Pour ce faire, une partie de la suite de bits représentant notre instruction à éxecuter va stocker cette adresse.



Dans ce cas, ces processeurs utilisent toujours un registre pour stocker cette adresse, mais ne possèdent pas de compteur ordinal, et n'ont pas besoin de calculer une adresse qui leur est fournie sur un plateau.

Et que ca saute !

En gros, un processeur est sacrément stupide s'il ne fait qu'exécuter des instructions dans l'ordre. Certains processeurs ne savent pas faire autre chose, comme le Harvard Mark I, et il est difficile, voire impossible, de coder certains programmes sur de tels ordinateurs. Mais rassurez-vous : on peut faire mieux ! Il existe un moyen permettant au processeur de faire des choses plus évoluées et de pouvoir plus ou moins s'adapter aux circonstances au lieu de réagir machinalement. Pour rendre notre ordinateur "plus intelligent", on peut par exemple souhaiter que celui-ci n'exécute une suite d'instructions que si une certaine condition est remplie. Ou faire mieux : on peut demander à notre ordinateur de répéter une suite d'instructions tant qu'une condition bien définie est respectée.

Diverses **structures de contrôle** de ce type ont donc étés inventées. Voici les plus utilisées et les plus courantes : ce sont celles qui reviennent de façon récurrente dans un grand nombre de langages de programmation actuels. On peut bien sur en inventer d'autres, en spécialisant certaines structures de contrôle à des cas un peu plus particuliers ou en combinant plusieurs de ces structures de contrôles de base, mais cela dépasse le cadre de ce tutoriel : ce tutoriel ne va pas vous apprendre à programmer.

Nom de la structure de contrôle	Ce qu'elle fait
SI...ALORS	exécute une suite d'instructions si une condition est respectée
SI...ALORS...SINON	exécute une suite d'instructions si une condition est respectée ou exécute une autre suite d'instructions si elle ne l'est pas.
Boucle ***WHILE...DO***	répète une suite d'instructions tant qu'une condition est respectée.
Boucle ***DO...WHILE*** aussi appelée ***REPEAT UNTIL***	répète une suite d'instructions tant qu'une condition est respectée. La différence, c'est que la boucle ***DO...WHILE*** exécute au moins une fois cette suite d'instructions.
Boucle ***FOR***	répète un nombre fixé de fois une suite d'instructions.

Concevoir un programme (dans certaines langages de programmation), c'est simplement créer une suite d'instructions, et utiliser ces fameuses structures de contrôle pour l'organiser. D'ailleurs, ceux qui savent déjà programmer auront reconnu ces fameuses structures de contrôle.

Reste à adapter notre ordinateur pour que celui-ci puisse supporter ces fameuses structures. Et pour cela, il suffit simplement de rajouter quelques instructions.
Pour que notre ordinateur puisse exécuter ces structures de contrôle basiques, il utilise :

- des **instructions de test**, qui vérifient si une condition est respectée ou pas ;
- et les instructions de saut ou **branchements**.

Instructions de test

Ces instructions sont des instructions assez simples : elles permettent de comparer deux valeurs. Elles permettent généralement de comparer des nombres entiers. Mais certains processeurs fournissent des instructions pour comparer des nombres à virgule flottante, des caractères, ou des données plus complexes.

Ces instructions permettent souvent d'effectuer les comparaisons suivantes :

- A < B (est-ce que A est supérieur à B ?) ;
- A > B (est-ce que A est inférieur à B ?) ;
- A == B (est-ce que A est égal à B ?) ;
- A != B (est-ce que A est différent de B ?.

Ces instructions dites de test ou de comparaison vont chacune fournir un résultat qui est codé sur un bit, qui permettra de dire si la condition testée est vraie ou fausse. Dans la majorité des cas, ce bit vaut 1 si la condition testée est vraie, et 0 sinon. Dans de rares cas, c'est l'inverse.

Bien sur, ce bit de résultat n'est pas placé n'importe où : notre processeur incorpore un registre spécialisé, qui stocke ces résultats des comparaisons, ces bits. Il s'agit du **registre d'état**. Ce registre d'état est un registre, qui ne stocke pas de données comme des nombres entiers, ou des nombres flottants, mais stocke des bits individuels. Chacun de ces bits a une signification propre, prédéterminée lors de la conception du processeur. Le bit du registre d'état qui est modifié par une instruction de test dépendent de l'instruction utilisée. Par exemple, on peut utiliser un bit qui indiquera si l'entier testé est égal à un autre, un autre bit qui indiquera si le premier entier testé est supérieur égal à l'autre, etc.

Il arrive que certaines de ces instructions de test effectuent plusieurs comparaisons en même temps et fournissent plusieurs résultats : l'instruction de test modifie plusieurs bits du registre d'état en une seule fois. Par exemple, un processeur peut très bien posséder une instruction **cmp** capable de vérifier si deux valeurs A et B sont égales, différentes, si A est inférieure à B, et si A est supérieur à B, en même temps.

l'exception qui confirme la règle

Ces instructions de test peuvent parfois être remplacées par des instructions arithmétiques. Ce registre d'état ne sert pas que pour les comparaisons. Les bits qu'il contient peuvent servir à autre chose : on peut utiliser un bit pour indiquer si le résultat d'une opération arithmétique est égal ou non à zéro, un autre pour indiquer si ce résultat est négatif ou non, encore un autre pour savoir si le résultat est pair ou impair, etc. Et pour cela, certaines instructions arithmétiques vont modifier ces bits du registre d'état. Ainsi, nos instructions arithmétiques peuvent ainsi remplacer une instruction de test : elles positionnant des bits du registre d'état, suivant une condition (résultat égal à zéro, négatif, etc).

Par exemple, prenons le cas d'un processeur qui

- possède une instruction machine permettant d'effectuer une soustraction ;
- et qui possède un bit nommé *NULL* dans le registre d'état qui vaut 1 si le résultat d'une opération est nul, et qui vaut 0 sinon.

Il est parfaitement possible que l'instruction de soustraction puisse mettre à jour ce bit *NULL*. Dans ce cas, on peut tester si deux nombres sont égaux en soustrayant leur contenu : si les deux registres contiennent la même valeur, le bit *Null* correspondant au registre dans lequel on stocke le résultat sera positionné à 1 et on saura si la condition est remplie.

D'autres bits peuvent être modifiés suivant le signe du résultat ou d'autres facteurs (une éventuelle retenue, par exemple). Cela permet de déterminer si la valeur soustraite est supérieure ou inférieure à la valeur à l'autre valeur. D'ailleurs, sur certains processeurs, l'instruction **cmp** (mentionnée plus haut) n'est qu'une soustraction déguisée dans un opcode différent (il faut aussi préciser que le résultat de la soustraction n'est pas sauvegardé dans un registre ou en mémoire et est simplement perdu). C'est le cas sur certains processeurs ARM ou sur les processeurs x86.

Branchements

Quoiqu'il en soit, ces instructions de test ne permettent pas de créer des structures de contrôle à elles seules. Elles doivent impérativement être combinées avec d'autres instructions spéciales, qu'on appelle des branchements. Ces **branchements** sont des instructions qui modifient la valeur stockée dans le registre d'adresse d'instruction.

Elles permettent de sauter directement à une instruction autre que l'instruction immédiatement suivante et poursuivre l'exécution à partir de cette instruction. Cela permet au programme de passer directement à une instruction située plus loin dans le déroulement normal du programme, voire de revenir à une instruction antérieure, plutôt que d'exécuter les instructions dans l'ordre.

Branchements conditionnels et inconditionnels

Il existe deux types de branchements.

- **Les branchements inconditionnels** : le processeur passe toujours à l'instruction vers laquelle le branchement va renvoyer.
- **Les branchements conditionnels**. L'instruction de branchement n'est exécutée que si certains bits du registre d'état sont à une certaine valeur (qui peut aussi bien 0 que 1 suivant l'instruction de branchement utilisée).

Les branchements conditionnels sont souvent précédés d'une instruction de test ou de comparaison qui va modifier un ou plusieurs des bits du registre d'état. C'est souvent ainsi qu'on va fabriquer nos structures de contrôle.

Sur certains processeurs, certains branchements conditionnels sont un peu plus malins : ils effectuent le test et le branchement en une seule instruction machine. Si on réfléchit bien, les instructions de test sont presque toujours suivies d'un branchement conditionnel. Pareil pour les branchements conditionnels, qui sont presque toujours précédés d'une instruction de test. Dans ce cas, autant utiliser une seule instruction qui effectue le test et le branchement ! Cela permet de se passer des instructions de test et du registre d'état.

A l'inverse, certains processeurs sont un peu plus extrêmes, mais dans l'autre sens : ils se passent totalement de branchements conditionnels et ils émulent d'une façon assez particulière. Sur ces processeurs, les instructions de test sont un peu spéciales : si la condition testée par l'instruction est fausse, elle vont simplement zapper l'instruction immédiatement suivante. On peut ainsi créer l'équivalent d'un branchement conditionnel avec une instruction de test suivie d'un branchement inconditionnel.

Chacune de ces trois méthodes a ses avantages et ses inconvénients, qu'on abordera pas ici. Je peux juste vous dire que la méthode fusionnant les instructions de tests et de branchement permet de se passer de registre d'état et d'avoir un programme plus petit (on utilise une seule instruction au lieu de deux pour chaque branchement). Par contre, la méthode avec instructions de tests et branchements conditionnels séparés permet de mieux gérer des *overflows* ou certaines conditions plus efficacement : une instruction arithmétique peut positionner un bit du registre d'état pour guider un branchement (on l'a vu dans le paragraphe "l'exception qui confirme la règle"). Mais je m'arrêterais là : sachez juste qu'il existe d'autres avantages et désavantages sur les processeurs modernes.

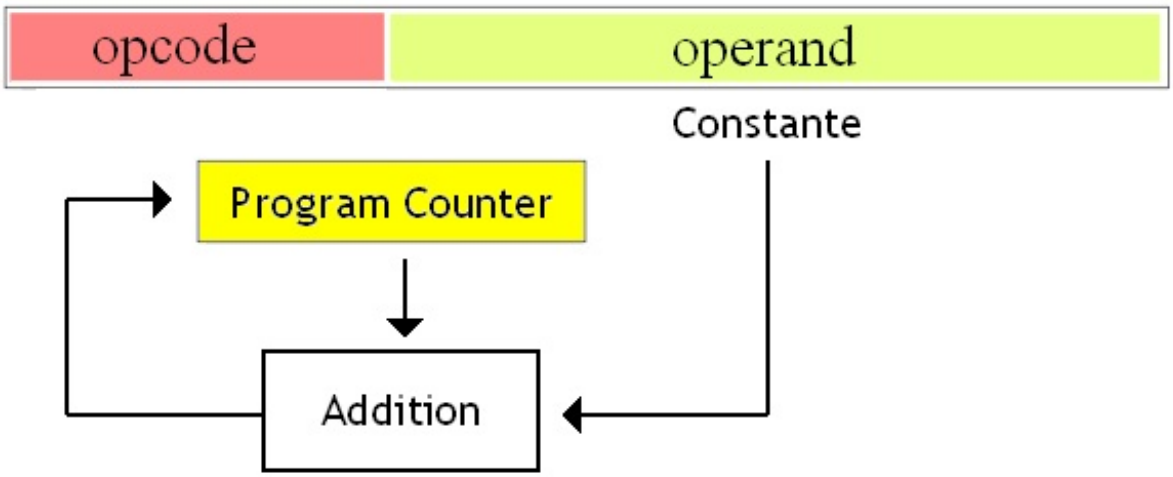
Modes d'adressage

Ces branchements peuvent avoir trois modes d'adressages :

- direct,
- par offset,
- indirect à registre.

Dans le premier cas, l'opérande est simplement l'adresse de l'instruction à laquelle on souhaite reprendre.

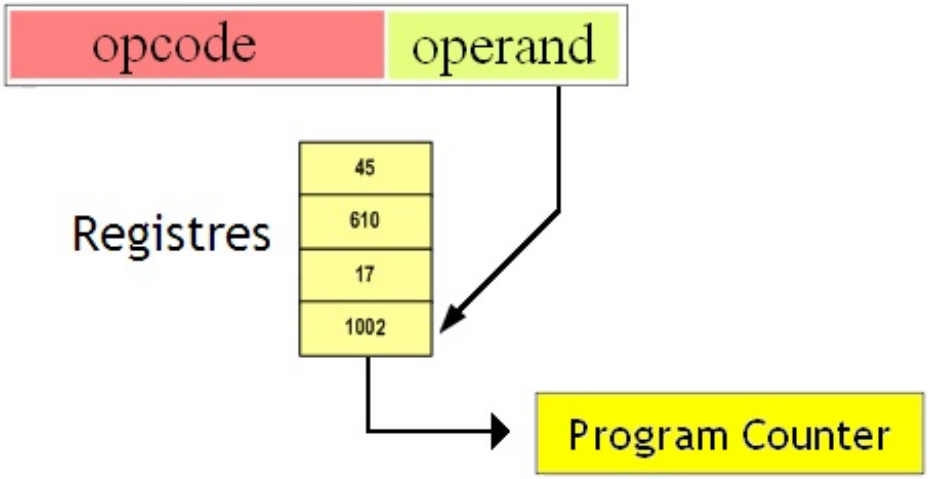
Dans le second cas, l'opérande est un nombre. Il suffit d'ajouter ce nombre à l'adresse déjà présente dans le registre d'adresse d'instruction pour tomber sur l'adresse voulue. On appelle ce nombre un *Offset*. De tels branchements sont appelés des **branchements relatifs**.



Les branchements basés sur des décalages permettent de localiser un branchement par rapport à l'instruction en cours d'exécution : par exemple, cela permet de dire "le branchement est 50 instructions plus loin". Cela facilite la création de programmes qui se moquent complètement de l'adresse à laquelle on les charge en mémoire. Sur certains ordinateurs capables d'exécuter plusieurs programmes "en même temps", un programme n'est presque jamais chargé en mémoire au même endroit et les adresses changent à chaque exécution.

Par défaut, les branchements qui ne sont pas des branchements relatifs considèrent que le programme commence à l'adresse zéro et localisent leurs instructions à partir de la première adresse d'un programme. Pour localiser l'adresse de destination, on est obligé de la calculer à l'exécution du programme, à partir de l'adresse de base du programme, en effectuant une addition. Les branchements relatifs ne sont pas concernés par ce genre de problème : ceux-ci ne repèrent pas l'adresse de destination du branchement par rapport à l'adresse à laquelle est placée le branchement, contenue dans le registre d'adresse d'instruction.

Il existe un dernier mode d'adressage pour les branchements : l'adresse vers laquelle on veut brancher est stockée dans un registre. L'opérande du branchement est donc un registre. Avec de tels branchements, l'adresse vers laquelle on souhaite brancher peut varier au cours de l'exécution du programme. On appelle de tels branchements des **branchements indirects**.



Ces branchements indirects sont à opposer aux autre branchements qui sont ce qu'on appelle des **branchements directs** : avec ces derniers, l'adresse vers laquelle il faut brancher est constante et ne peut en aucun cas varier (sauf cas particuliers utilisant du *Self-modifying code*). Les branchemetns directs sont souvent utilisés pour créer les structures de contrôle dont j'ai parlé plus haut. Par contre, les branchements indirects sont souvent camouflés dans des fonctionnalités un peu plus complexes de nos langages de programmation (pointeurs sur fonction, chargement dynamique de DLL, structure de contrôle *Switch*, etc) et il n'est pas si rare d'en utiliser sans en avoir conscience.

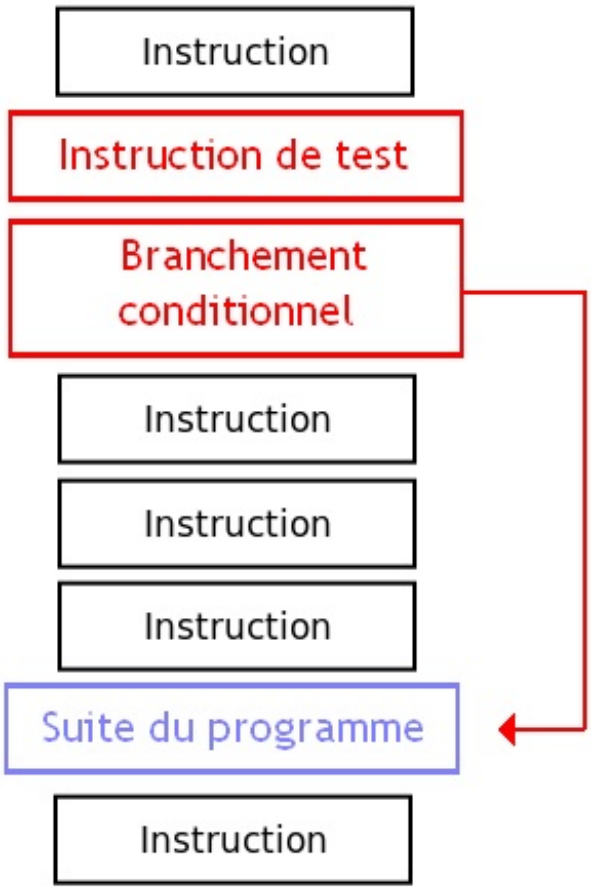
Structures de contrôle, tests et boucles

Maintenant que l'on a appris ce qu'étaient les branchements et les tests, on va voir comment faire pour fabriquer nos fameuses structures de contrôle avec. Bien sûr, on ne va pas aller très loin : juste voir ce qu'il faut pour comprendre la suite du tutoriel. Par exemple, ce qu'on va voir sera assez utile lorsqu'on étudiera la prédiction de branchement.

Le *Si...Alors*

Implémenter un simple *Si...Alors* est assez intuitif : il suffit de tester la condition, et de prendre une décision suivant le résultat du branchement.

Voici ce que cela peut donner :



Avec cette organisation, on utilise un branchement tel que si la condition testée soie vraie, alors on poursuit l'exécution sans reprendre à la suite du programme : on exécute alors la suite d'instruction du *Si...Alors*, avant de reprendre à la suite du programme. Dans le cas où la condition testée est fausse, on reprend directement à la suite du programme, sans éxecuter la suite d'instruction du *Si...Alors*.

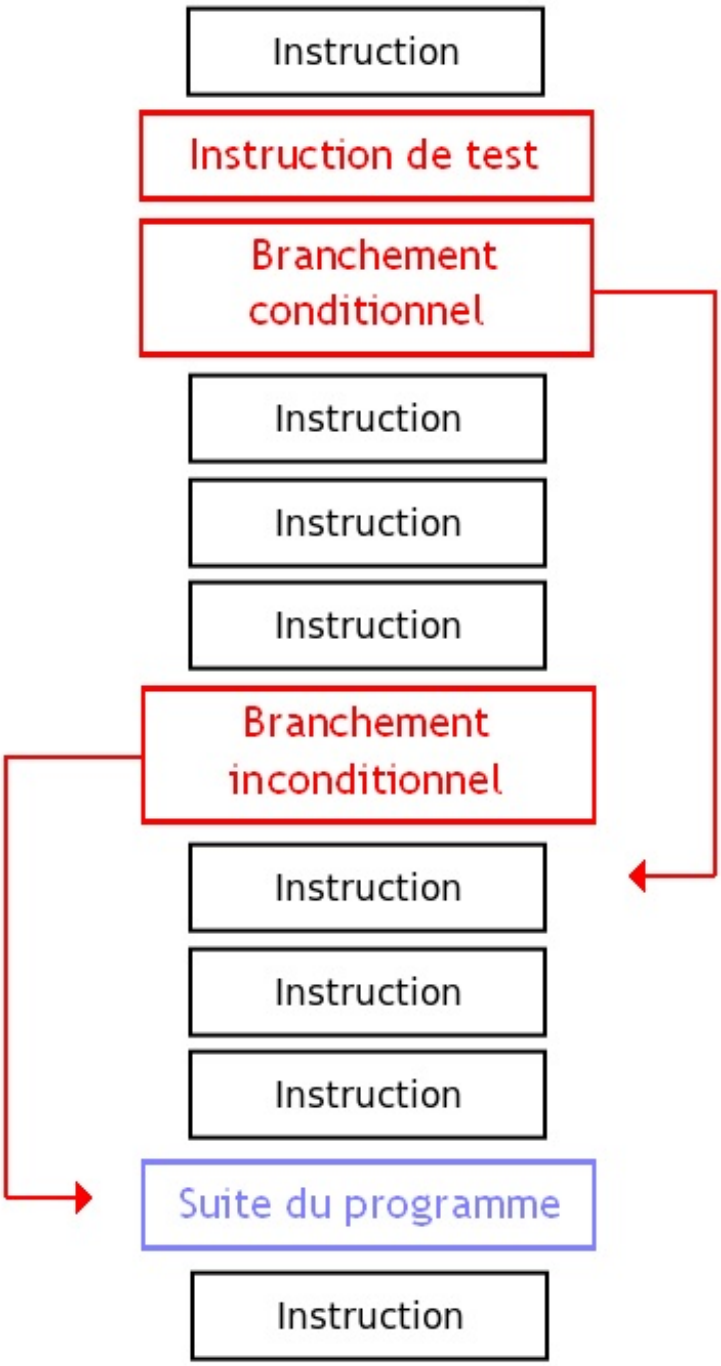
Et ça marche ! Enfin presque, i y a un problème : comment faire pour que le branchement conditionnel puisse faire ce qu'on lui demande ? Et oui, j'ai dit plus haut qu'un branchement est exécutée quand la condition testée est vraie, valide, et pas autrement.

Pour cela il y a, deux solutions. La première solution est assez intuitive : il suffit de tester la condition inverse à celle nécessaire pour faire exécuter notre suite d'instruction de la structure de contrôle *Si...Alors*. Pour la seconde solution, quelques explications s'imposent. Comme je l'ai dit plus haut, notre branchement va s'exécuter si certains bits du registre d'état sont placés à une certaine valeur. Sur certains processeurs, il peut exister des branchements différents qui vérifient le même bit du registre d'état. Seul différence : ces branchements s'exécutent pour des valeurs différentes de ce bit. Ainsi, pour chaque test possible, il existe un branchement qui s'exécute quand une condition est valide, et un autre qui s'exécute quand la condition n'est pas valide. Il suffit de choisir un branchement qui s'exécute quand une condition testée n'est pas valide pour éviter d'avoir à inverser le test.

Si...Alors...Sinon

Cette structure de contrôle sert à effectuer un traitement différent selon que la condition est respectée ou non : si la condition est respectée, on effectue une suite d'instruction ; tandis que si elle ne l'est pas, on effectue une autre suite d'instruction différente. C'est une sorte de *Si...Alors* contenant un second cas. Pour implémenter cette structure de contrôle, on peut donc améliorer le *Si...Alors* vu plus haut. La seule différence, c'est l'endroit vers lequel le branchement conditionnel va nous envoyer.

L'astuce est de placer les deux suites d'instructions les unes après les autres. Le branchement conditionnel enverra sur la suite à exécuter quand la condition voulue par le programmeur n'est pas réalisée. Dans le cas contraire, on poursuit l'exécution du programme après le branchement et on retombe sur le cas du *Si...Alors*.



Pour éviter d'exécuter les deux suites les unes après les autres, on place un branchement inconditionnel à la fin de la suite d'instruction à exécuter quand la solution est vraie. Ce branchement renverra sur la suite du programme.

Je tiens à signaler qu'on est pas limité à seulement deux suites d'instructions : on peut imbriquer des structures de contrôle *Si...Alors...Sinon* les unes dans les autres, si le besoin s'en fait sentir. Dans ce cas, il faut bien penser à mettre un branchement inconditionnel branchant sur la suite du programme après chaque suite d'instruction (sauf la dernière, qui est immédiatement suivie par la suite du programme).

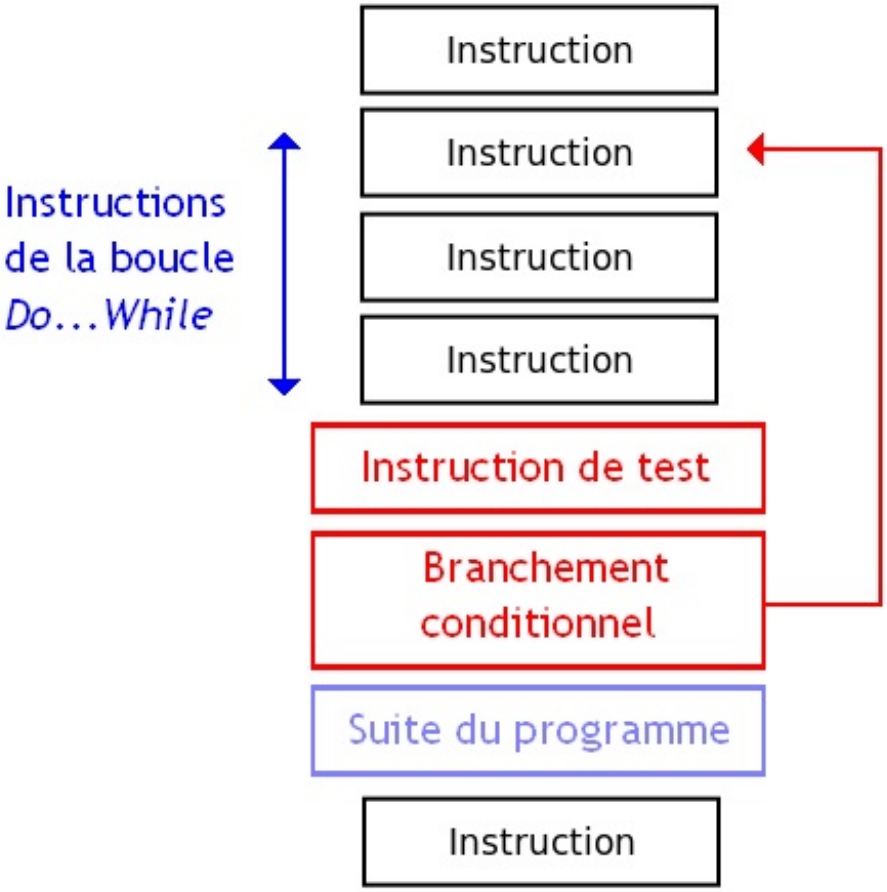
Boucles

Voyons maintenant comment un CPU fait pour exécuter une boucle. Pour rappel, **une boucle consiste à répéter une suite d'instructions CPU tant qu'une condition est valide (ou fausse)**. En gros, les boucles peuvent être vues comme des variantes de la structure de contrôle *Si...Alors*. Plutôt que de reprendre l'exécution du programme après la suite d'instructions exécutée quand la condition est valide (cas du *Si...Alors*), on reprend l'exécution du programme à l'instruction de test. Histoire de répéter les instructions déjà exécutées.

Boucle Do...While

Je vais tout d'abord commencer par la boucle la plus simple qui soit : la boucle *Do...While*. Dans la boucle *Do...While*, on souhaite que la suite d'instructions soit répétée tant qu'une certaine condition est vérifiée, et qu'en plus, elle soit exécutée au moins une fois !

Pour cela, la suite d'instructions à exécuter est placée avant les instructions de test et de branchement.

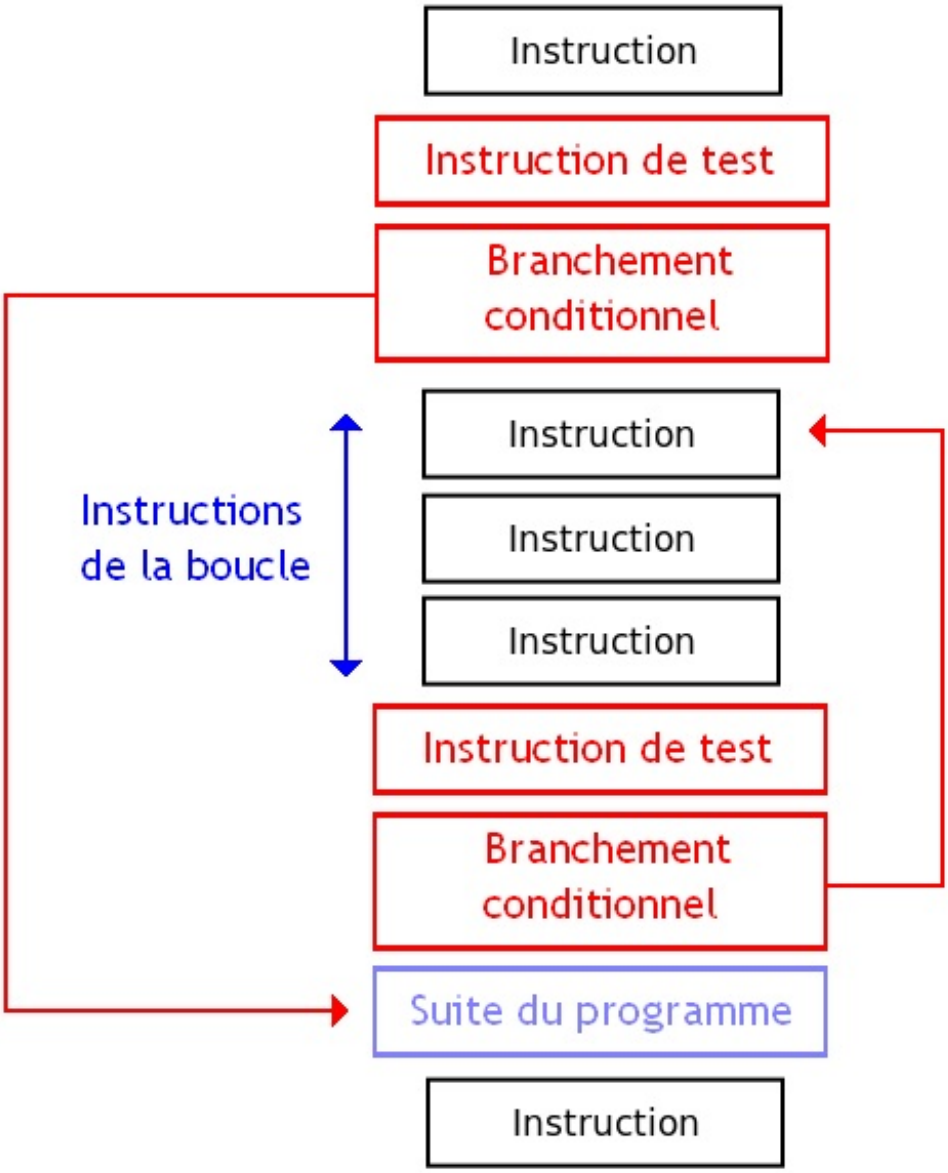


En faisant ainsi, notre suite d'instruction est exécutée une fois, avant de rencontrer une instruction de test qui va tester la condition de la boucle. Si jamais la condition testée est vérifiée (valide), alors notre branchement va renvoyer notre processeur vers la première instruction de la suite d'instruction à répéter. Si jamais la condition testée est fausse, on passe tout simplement à la suite du programme. Si on regarde bien, c'est bien le comportement qu'on attend d'une boucle *Do...While*.

While...Do, le retour !

Une boucle *While...Do* est identique à une boucle *Do...While* à un détail prêt : la suite d'instruction de la boucle n'a pas forcément besoin d'être exécutée. On peut très bien ne pas rentrer dans la boucle dans certaines situations.

On peut donc adapter une boucle *Do...While* pour en faire une boucle *While...Do* : il suffit de tester si notre boucle doit être exécutée au moins une fois, et exécuter une boucle *Do...While* équivalente si c'est le cas. Si la boucle ne doit pas être exécutée, on poursuit à partir de la suite du programme directement. Et pour cela, il suffit de placer une instruction qui teste si on doit renter dans la boucle, accouplée à un branchement qui renvoie sur la suite du programme si jamais on n rentre pas dans la boucle.



Boucle FOR

Une boucle *For* n'est qu'une boucle While...Do un peu spéciale. Aucune différence particulière dans la façon de créer une boucle for et une boucle *While...Do* n'existe dans la façon dont notre ordinateur exécute la boucle : c'est seulement ce qui est testé qui change et rien d'autre.

Sous-programmes : c'est fait en quoi une fonction ?

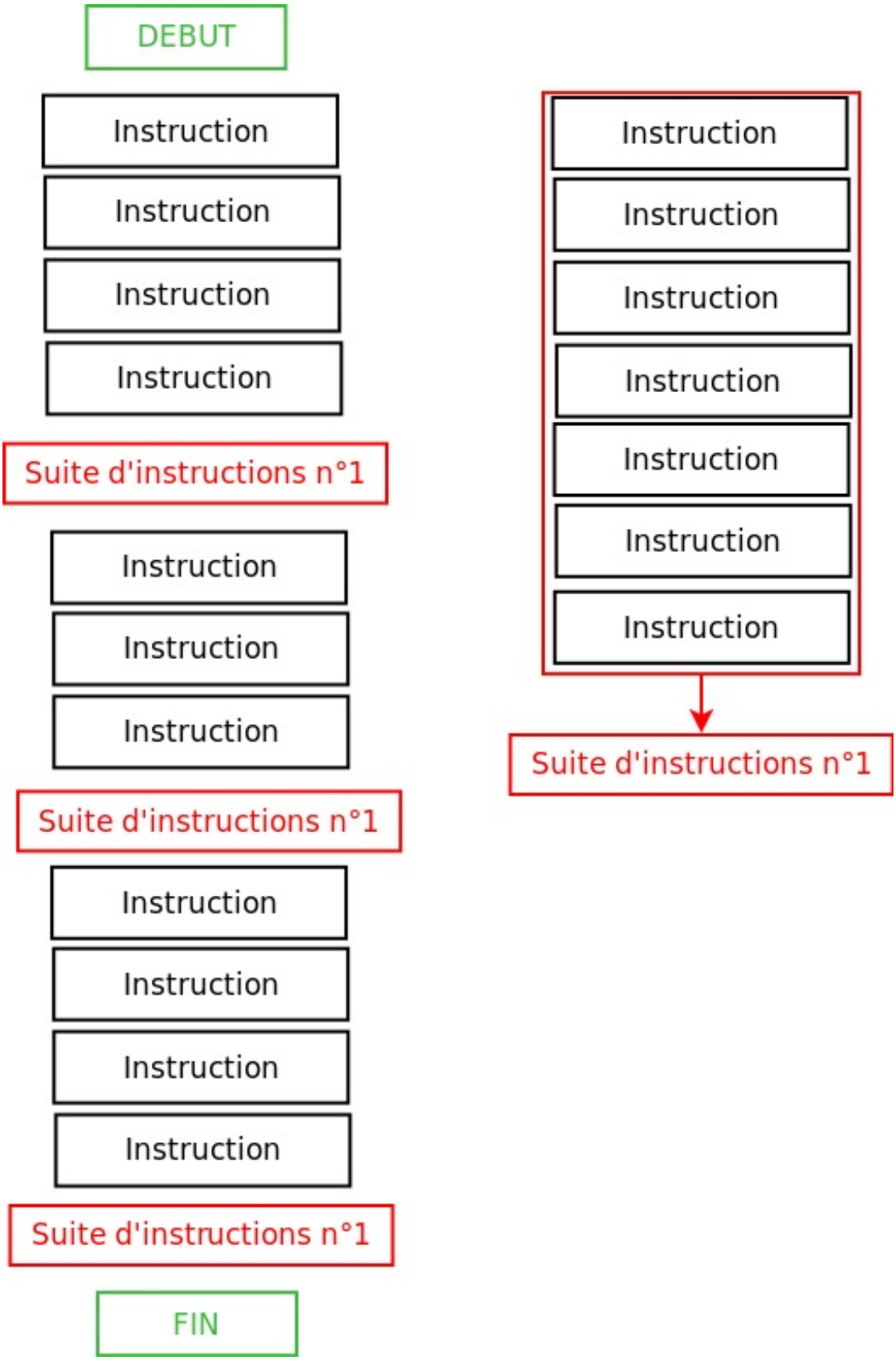
Ceux qui ont déjà faits de la programmation ont déjà certainement vus le concept de fonctions, quelque soit le langage qu'ils aient vus. On va ici expliquer ce qu'est une fonction, du point de vue du processeur ! Mais tout d'abord, on va clarifier un point de vocabulaire : on ne dira plus fonction, mais **sous-programme** ! Pour expliquer ce qu'est un sous-programme, il faut d'abord savoir à quoi ça sert. Car les sous-programmes sont des inventions qui répondent à des besoins bien précis : économiser de la mémoire et grandement faciliter la vie du programmeur !

A quoi ça sert ?

Lorsque vous créez un programme, le résultat sera une grosse suite d'instructions codées en langage machine, avec des 0 et des 1 partout : c'est une horreur ! Et parmi cette gigantesque suite d'instructions, il y a souvent des "sous-suites", des paquets d'instructions qui reviennent régulièrement et qui sont présents en plusieurs exemplaires dans le programme final. Ces sous-suites servent quasiment toujours à exécuter une tache bien précise et ont presque toujours une signification importante pour le programmeur. Par exemple, il va exister une de ces sous-suite qui va servir à calculer un résultat bien précis, communiquer avec un périphérique, écrire un fichier sur le disque dur, ou autre chose encore.

Sans sous-programmes

Sans utiliser de sous-programmes, ces suites d'instructions sont présentes en plusieurs exemplaires dans le programme. Le programmeur doit donc recopier à chaque fois ces suites d'instructions, ce qui ne lui facilite pas la tache (sauf en utilisant l'ancêtre des sous-programmes : les macros). Et dans certains programmes, devoir recopier plusieurs fois la séquence d'instruction qui permet d'agir sur un périphérique ou de faire une action spéciale est franchement barbant ! De plus, ces suites d'instructions sont présentes plusieurs fois dans le programme final, exécuté par l'ordinateur. Et elles prennent de la place inutilement !

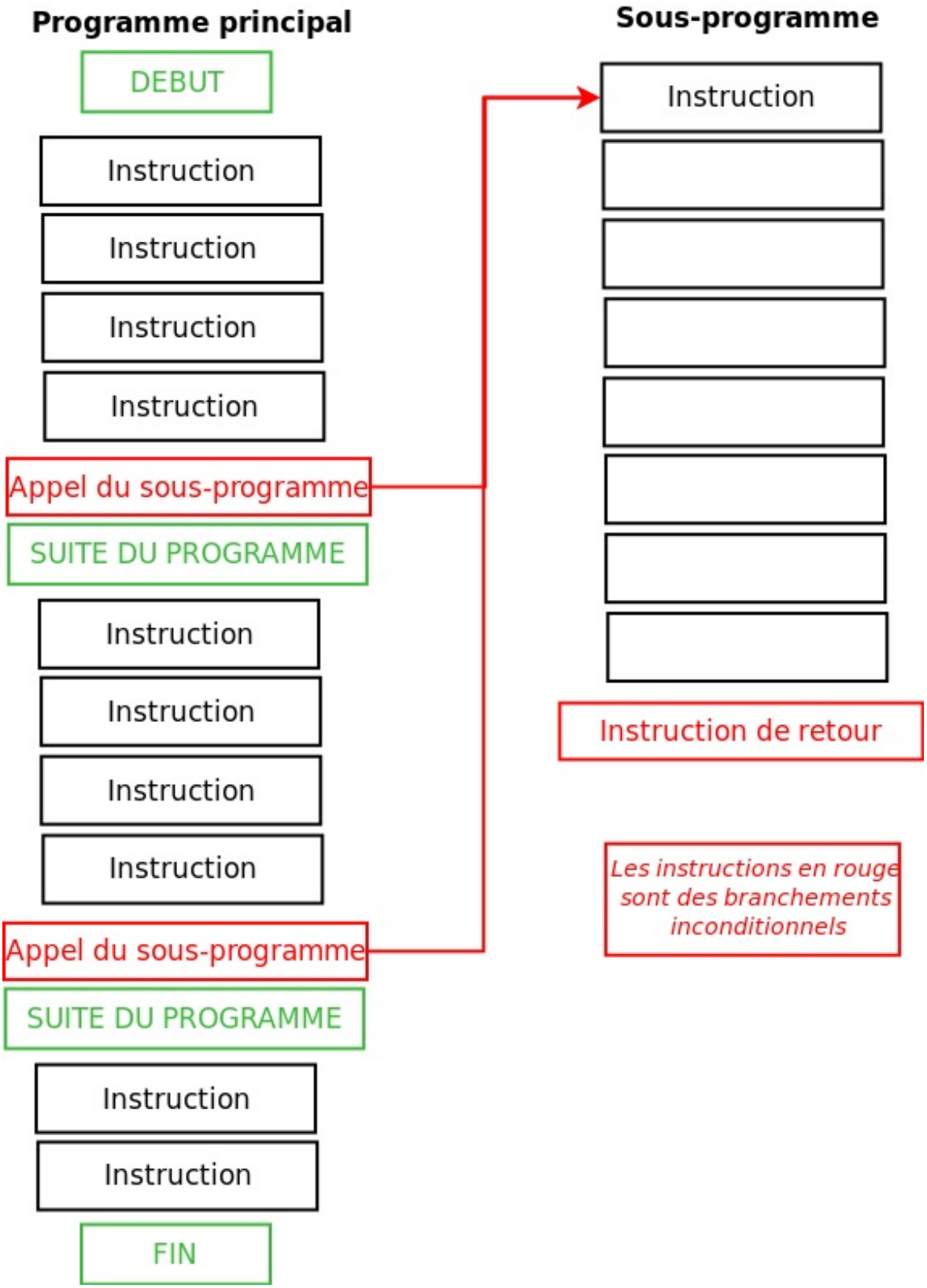


On a donc cherché un moyen qui ferait que ces suites d'instructions soient présentes une seule fois dans le programme et utilisables au besoin. On a donc inventé les **sous-programmes**.

Avec les sous-programmes

La technique du sous-programme consiste à ne mettre qu'un seul exemplaire de ces suites d'instructions. Cet exemplaire placé dans la mémoire, un peu à coté du programme "principal". On appellera cette suite d'instruction un **sous-programme**. En effet, un programme est une suite d'instruction. Donc, cette suite d'instructions peut être vue comme étant un programme que le programme principal exécutera au besoin. C'est au programmeur de "sélectionner" ces suites d'instructions qui apparaissent de façon répétée dans le programme, et d'en faire des sous-programmes.

Lorsqu'on a besoin d'exécuter ce sous-programme, il suffira d'exécuter une instruction de branchement qui pointera vers la première instruction de ce sous-programme. On dit alors qu'on appelle le sous-programme.



Retour vers la future (instruction) !

Logiquement, cette suite d'instructions apparait plusieurs fois dans notre programme. Comment savoir à quelle instruction reprendre l'exécution de notre programme, une fois notre sous-programme terminé ?

La solution : sauvegarder l'adresse de l'instruction à laquelle il faut reprendre ! Notre programme doit donc reprendre à l'instruction qui est juste après le branchement qui pointe vers notre sous-programme. Pour cela, on doit sauvegarder cette adresse appelée l'**adresse de retour**.
Cette sauvegarde peut être faite de deux manières :

- soit le processeur possède une instruction spéciale, capable de sauvegarder l'adresse de retour et de brancher vers le sous-programme voulu en une seule instruction ;
- soit on doit émuler cette instruction avec une instruction qui sauvegarde l'adresse de retour, suivie d'un branchement inconditionnel qui pointera vers notre sous-programme.

Pour le premier cas, cette instruction spéciale est capable de sauvegarder automatiquement l'adresse de retour et de brancher vers le sous-programme. On appelle cette instruction une **instruction d'appel de fonction**.

Reprenons là où on en était

Une fois le sous-programme fini, il suffit de charger l'adresse de retour dans le registre pointeur d'instruction pour reprendre l'exécution de notre programme principal là où il s'était arrêté pour laisser la place au sous-programme. Là encore, deux solutions sont possibles pour faire cela.

Sur certains processeurs, cela est fait par l'instruction située à la fin du sous-programme, qu'on nomme **instruction de retour**. C'est un branchement inconditionnel. Cette instruction a pour mode d'adressage, l'adressage implicite (l'adresse vers laquelle brancher est placée au sommet de la pile, pas besoin de la préciser).

Sur d'autres, cette instruction spéciale n'existe pas et il faut encore une fois l'émuler avec les moyens du bord. L'astuce consiste souvent à charger l'adresse de retour dans un registre et utiliser un branchement inconditionnel vers cette adresse.

Pile de sauvegarde des adresses de retour

Pour pouvoir exécuter plusieurs sous-programmes imbriqués (un sous-programme contient dans sa suite d'instructions un branchement vers un autre sous-programme), on permet de sauvegarder plusieurs adresses de retour : une par sous-programme. À chaque fin de sous-programme, on est obligé de choisir quelle est la bonne adresse de retour parmi toutes celles qui ont étés sauvegardées.

Pour cela, on a encore une fois deux solutions différentes :

- soit on stocke des adresses de retour dans chaque Stack frame ;
- soit on les stocke dans les registres.

Avec la première solution, les adresses de retour sont stockées dans la pile. Lorsque l'on appelle un sous-programme, l'adresse de retour est sauvegardée au sommet de la pile, au-dessus de toute les autres. Ce stockage des adresses de retour utilisant la pile permet de toujours retourner à la bonne adresse de retour. Bien sûr, il y a une limite aux nombres de *stack frame* qu'on peut créer dans la pile. Et donc, un nombre maximal de sous-programmes imbriqués les uns dans les autres.

Néanmoins, certains processeurs ne peuvent pas gérer de pile en mémoire RAM, et doivent donc trouver un moyen d'émuler cette pile d'adresse de retour sans utiliser de pile. La solution, est d'incorporer dans le processeur des registres spécialisés, organisés en pile, dont le but est de conserver l'adresse de retour. Ainsi, à chaque appel de sous-programme, l'adresse de retour sera stockée dans un de ces registres pour être conservée.

Certains processeurs utilisent un mélange des deux solutions : une partie des adresses de retour (celles sauvegardées le plus récemment) est conservée dans les registres du processeur pour plus de rapidité, et le reste est sauvegardé dans la pile.

Paramètres et arguments

Notre sous-programme va parfois modifier des données en mémoire RAM. Cela peut poser problème dans certains cas : on peut avoir besoin de conserver les anciennes données manipulées par le sous-programme. Pour cela, le sous-programme va devoir

manipuler une copie de ces données pour que notre sous-programme puisse les manipuler à loisir. Ces copies seront appelées des **arguments** ou encore des **paramètres**. Ces paramètres sont choisis par le programmeur qui crée le sous-programme en question.

Pour cela deux solutions :

- soit on les passe par la pile ;
- soit on passe les arguments dans les registres.

La première solution utilise la pile. Vu que notre pile est une simple portion de la mémoire RAM, on peut stocker ce qu'on veut dans une *stack frame*, et pas seulement une adresse de retour. On va donc, dans certains cas, copier les données à manipuler dans la pile. C'est ce qu'on appelle le **passage par la pile**.

Seconde solution : stocker directement les arguments dans les registres du processeur si ils sont peu nombreux, sans avoir à utiliser la pile. Comme cela, le processeur pourra les manipuler directement sans devoir les charger depuis la RAM ou les stocker dans le cache. C'est le **passage par registre**.

Suivant le langage de programmation, le compilateur, le système d'exploitation ou le processeur utilisé, les sous-programmes utiliseront soit le passage par la pile, soit le passage par registre, ou encore un mélange des deux. Généralement, le passage par la pile est très utilisé sur les processeurs CISC, qui ont peu de registres. Par contre, les processeurs RISC privilégient le passage par les registres : il faut dire que les processeurs RISC ont souvent un grand nombre de registre, ce qui permet de passer beaucoup d'arguments sans trop de problèmes.

Une histoire de registres

Enfin, pour terminer, la pile va aussi servir à stocker le contenu de l'ensemble des registres du processeur tels qu'ils étaient avant qu'on exécute notre sous-programme. En effet, lorsqu'un sous-programme s'exécute, il va utiliser certains registres, qui sont souvent déjà utilisés par le programme.

Pour éviter de remplacer le contenu des registres par une valeur calculée-allouée par notre sous-programme, on doit donc conserver une copie de ceux-ci dans la pile. Une fois que le sous-programme a finit de s'exécuter, on remet les registres dans leur état original, en remettant leur sauvegarde depuis la pile, dans les registres adéquats. Ce qui fait que lorsqu'un sous-programme a fini son exécution, tous les registres du processeur sont reviennent à leur ancienne valeur : celle qu'ils avaient avant que le sous-programme ne s'exécute. Rien n'est effacé !

Influence du nombre de registres

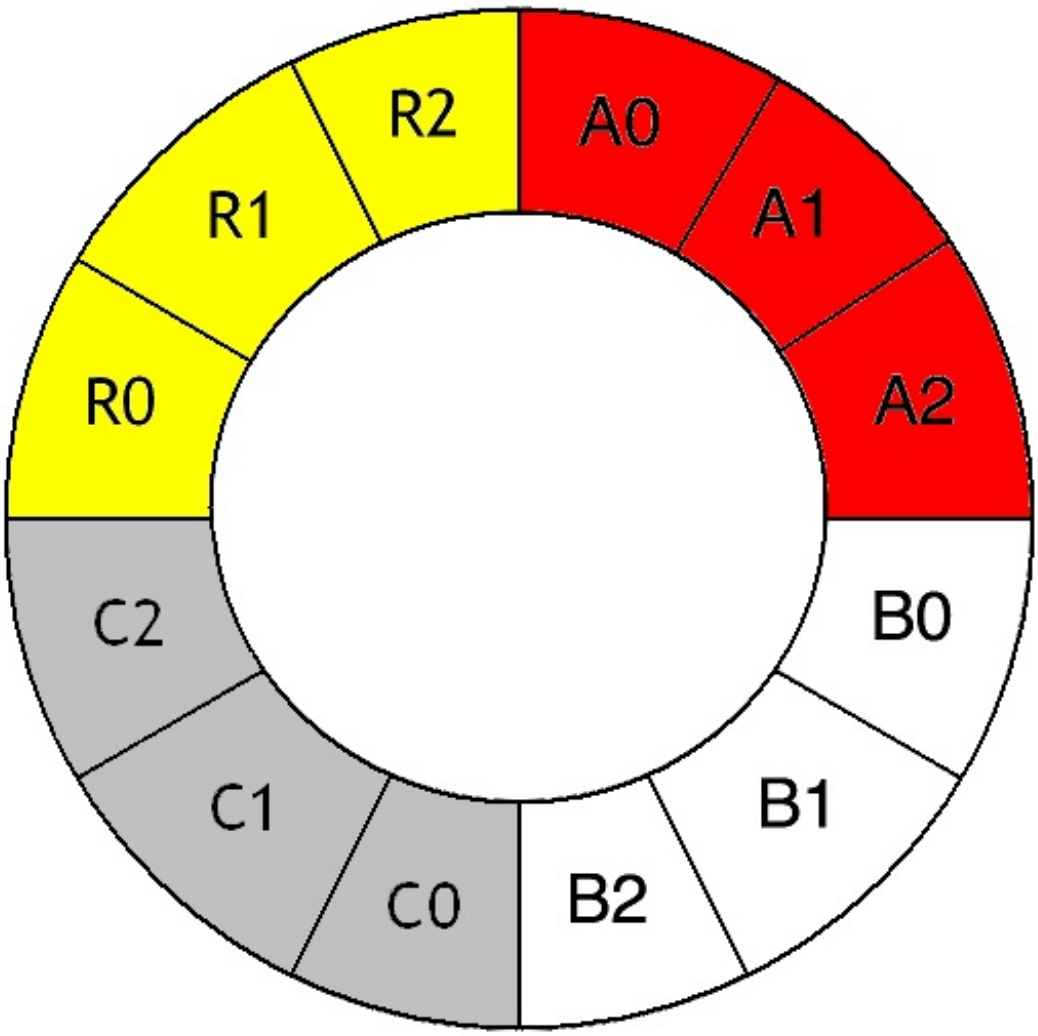
Bien évidemment, plus un processeur a de registres architecturaux (des registres qui ont un nom et sont manipulables par des instructions), plus cette sauvegarde de registre prend du temps. Si les processeurs CISC sont épargnés avec leur faible nombre de registres, ce n'est pas le cas des processeurs RISC. Ceux-ci contiennent un grand nombre de registres et sauvegarder ceux-ci prend du temps. Quand aux machines à piles pures, elles sont tranquilles : il n'y a pas de registres à sauvegarder, hormis le registre pointeur d'interruption. Autant vous dire que sur ces machines à pile, les appels de sous-programmes sont supers rapides.

Fenêtrage de registres

Quoiqu'il en soit, exécuter un sous-programme prend du temps : il faut sauvegarder l'adresse de retour, empiler les arguments, empiler une copie des registres du processeur, etc. Pour limiter le temps mit à gérer la sauvegarde des registres, certains processeurs utilisent une petite astuce assez sympa nommée le **fenêtrage de registres**.

Les processeurs utilisant le fenêtrage de registres possèdent des registres supplémentaires cachés, invisibles pour le programmeur. Par exemple, si notre processeur possède 8 registres architecturaux, (ce sont des registres qui possèdent un nom et qui sont donc manipulables par notre programme), alors le fenêtrage de registre impose la présence de 2, 3, 4, 8 fois plus de registres. Dans le cas du fenêtrage de registre, seule une petite partie de ces registres peut être manipulable par un programme ou un sous-programme : on appelle cette partie une **fenêtre de registre**. Ces fenêtres ont toute la même taille, et contiennent autant de registres qu'il y a de registres architecturaux.

Prenons un exemple : un processeur dont le jeu d'instruction contient 3 registres architecturaux, et qui contient en réalité 12 registres. Ces 12 registres sont regroupés dans des paquets, des fenêtres de 3 registres : cela nous fait donc 4 fenêtres différentes.



Lorsque notre programme s'exécute, il va utiliser 3 registres en tout : une fenêtre sera donc utilisé par notre programme principal. Si un sous-programme veut s'exécuter, notre processeur va faire en sorte que notre sous-programme utilise une fenêtre (un paquet) de registres inutilisée : pas besoin de sauvegarder les 3 registres principaux, vu que notre sous-programme ne les utilisera pas et manipulera 3 registres différents appartenant à une autre fenêtre !

Bien sûr, cela fonctionne pour chaque sous-programme qui cherche à s'exécuter : on lui attribue une fenêtre de registres vierge, inutilisée, qu'il pourra utiliser à loisir, au lieu d'utiliser une fenêtre déjà prise. Évidemment, cela ne marche que s'il reste une fenêtre inutilisée. Dans le cas contraire, on est obligé de sauvegarder les registres d'une fenêtre dans la pile. Avec notre exemple à 4 fenêtres, on peut donc exécuter un programme, et 3 sous-programmes imbriqués.

Au fait : vous avez remarqué ? Un sous-programme est censé manipuler des registres architecturaux, mais utilisera en fait des registres cachés, localisés dans une autre fenêtre. Le fait est que chaque registre architectural est présent en plusieurs exemplaires, un dans chaque fenêtre. Ainsi, deux registres placés dans des fenêtres différentes peuvent avoir le même nom. Lorsqu'un sous-programme s'exécute, il manipulera le registre architectural localisé dans la fenêtre qui lui a été attribuée. On voit bien que les registres utilisables en assembleur sont différents des registres réellement présent dans le processeur. Et ce n'est pas la seule situation dans laquelle certains registres architecturaux seront présents en double, comme on le verra dans la suite du tutoriel.

Valeur de retour

Un sous-programme peut parfois servir à calculer un résultat, et qu'il faut bien fournir ce résultat quelque part. On pourra donc le récupérer pour faire quelque chose avec. Généralement, c'est le programmeur qui décide de conserver une donnée. Celui-ci peut avoir besoin de conserver le résultat d'un calcul pour pouvoir l'utiliser plus tard, par exemple. Ce résultat dépend fortement du sous-programme, et peut être n'importe quelle donnée : nombre entier, nombre flottant, tableau, objet, ou pire encore. Cette donnée est appelée la **valeur de retour** du sous-programme.

Cette donnée, après avoir été "calculée" par le sous-programme, devra être conservée quelque part : calculer une valeur de retour

pour l'effacer serait stupide, non ? Dans certains langages (ou suivant le compilateur), tous les registres du processeur sont remis à leur valeur originelle lorsqu'un sous-programme se termine. Il va de soit que l'on ne peut pas stocker cette valeur de retour dans un registre : elle serait écrasée lors de la restauration des registres. Sans compter que cette valeur ne tient pas toujours dans un registre : un registre contenant 64 bits pourra difficilement contenir une valeur de retour de 5 kilo-octets.

Pour cela, deux solutions :

- soit on stocke ces valeurs de retour dans la pile ;
- soit on dédie certains registres à notre valeur de retour, et on se débrouille pour que la restauration des registres ne touche par ceux-ci.

Première solution : sauvegarder la valeur de retour sur la pile. Ainsi, la valeur de retour est présente au sommet de la pile et peut être utilisée si besoin.

J'ai dit plus haut que tous les registres du processeur sont restaurés lors du retour d'un sous-programme. Sauf que cela dépend des langages et des compilateurs : certains gèrent les registres de façon à ne pas remettre tous les registres à leur valeur d'origine immédiatement. Cela permet de conserver une valeur de retour dans les registres, dès que possible.

Suivant la taille de la valeur de retour, diverses méthodes sont envoyées pour conserver celle-ci.

- on peut conserver une valeur de retour quand elle est capable de tenir dans les registres du processeur : une valeur de retour peut prendre un ou plusieurs registres à elle toute seule, tout dépend de ce qu'on veut sauvegarder ;
- on peut aussi de sauvegarder la valeur de retour dans la mémoire RAM ou dans la pile, et conserver l'adresse de l'endroit dans lequel la valeur de retour est localisée en mémoire dans un registre.

On peut aussi inventer d'autres solutions (ne l'oubliez pas), mais les deux solutions citées plus haut sont les plus simples à expliquer.

Variables automatiques

Un sous-programme doit généralement manipuler des données temporaires qui lui permettent de faire ce qu'on lui demande. Ces données sont des données internes au sous-programme, que lui seul doit manipuler et qui ne doivent généralement pas être accessibles à d'autres programmes ou sous-programmes. Dans certains langages de programmation, on appelle ces variables des **variables locales**.

Une solution pour gérer ces variables pourrait consister à réserver une portion de la mémoire chaque sous-programme, spécialement dédiée au stockage de ces variables. Mais cela prendrait trop de mémoire, et réserver définitivement de la mémoire pour stocker des données temporaires de façon occasionnelle serait du gâchis. Sans compter le cas dans lequel un sous-programme s'appelle lui-même : il vaut mieux que ces appels de sous-programmes aient des données placées dans des emplacements mémoires bien séparés, pour éviter des catastrophes.

Pour cela, ces données sont crées pendant l'exécution du sous-programme, restent en mémoire tant que le sous-programme s'exécute, et sont effacées dès que le sous-programme termine. La solution consiste à réserver une partie d'une *stack frame* pour stocker ces données durant l'exécution du sous-programme. En effet, quand on dépile une *stack frame* à la fin de l'exécution d'un sous-programme, le contenu de celle-ci est complétement perdu. On appelle de telles données temporaires, stockées sur la pile des **variables automatiques**.

On peut aussi stocker ces données internes que le sous-programme doit manipuler dans les registres du processeur, si elles sont peu nombreuses. Comme cela, le processeur pourra les manipuler directement sans devoir les charger depuis la RAM ou les stocker dans le cache.

Pour manipuler ces variables automatiques, ainsi que les arguments/paramètres, notre processeur dispose parfois de modes d'adressages spécialisés, qui permettent de sélectionner une donnée dans une *Stack Frame*. Généralement, ces modes d'adressage permettent d'ajouter une constante à l'adresse du début de la *Stack Frame* (cette adresse étant stockée dans un registre).

Plusieurs piles

Certains processeurs ne possèdent qu'une seule pile dans laquelle un programme mettra à la fois les adresses de retour, les variables locales, les paramètres, et le reste des informations nécessaires pour exécuter notre sous-programme sans accros.

Néanmoins, certaines processeurs sont plus malins et possèdent deux ou plusieurs piles, chacune étant spécialisée.
On peut ainsi avoir :

- une pile pour le adresses de retour ;
- et une autre pour les paramètres et les variables locales.

Certains processeurs possèdent carrément trois piles :

- une pour les adresses de retour ;
- une pour les paramètres ;
- et une pour les variables locales.

Cela permet d'éviter de recopier les arguments ou les variables locales d'un sous-programme sur la pile pour les passer à un autre sous-programme si besoin est.
Et voilà, nous avons commencé à effleurer les différentes instructions et autres détails architecturaux qui ont permis d'adapter nos processeurs aux langages de haut niveau dits procéduraux. Alors certes, ça semble assez bas niveau et ne semble peut-être pas vraiment convainquant, mais le matériel peut aller beaucoup haut dans l'abstraction : il existe des processeurs capables de gérer nativement des langages fonctionnels ! Ce sont les architectures *dataflow*, que j'ai citées plus haut. Un bel exemple pour moi n'est autre que les machines LISP : ces machines possédaient de quoi exécuter nativement certaines primitives du langages LISP, un langage fonctionnel.

Comme autres exemples, certains processeurs implémentent directement dans leur circuits de quoi traiter la programmation objet : ils supportent l'héritage, des structures de données spéciales, des appels de données différents suivant que les fonctions appelées soient dans le même module/classe que l'appelant, et pleins d'autres choses. Comme quoi, l'imagination des constructeurs d'ordinateurs a de quoi surprendre ! Ne sous-estimez jamais les constructeurs d'ordinateurs.

Il y a quoi dans un processeur ?

Dans le chapitre sur le langage machine, on a vu notre processeur comme une espèce de boite noire contenant des registres qui exécutait des instructions les unes après les autres et pouvait accéder à la mémoire. Mais on n'a pas encore vu comment celui-ci était organisé et comment celui-ci fait pour exécuter une instruction. Pour cela, il va falloir nous attaquer à la **micro-architecture** de notre processeur. C'est le but de ce chapitre : montrer comment les grands circuits de notre processeur sont organisés et comment ceux-ci permettent d'exécuter une instruction. On verra que notre processeur est très organisé et est divisé en plusieurs grands circuits qui effectuent des fonctions différentes.

Execution d'une instruction

Le but d'un processeur, c'est d'exécuter une instruction. Cela nécessite de faire quelques manipulations assez spécifiques et qui sont toutes les mêmes quelque soit l'ordinateur. Pour exécuter une instruction, notre processeur va devoir faire son travail en effectuant des étapes bien précises.

Instruction Cycle

Il va d'abord devoir passer par une première étape : l'étape de ***Fetch***. Lors de cette première étape, le processeur va charger l'instruction depuis la mémoire et la stocker dans un registre spécialisé pour la manipuler. A la fin de cette étape, l'instruction est alors stockée dans un registre : le **registre d'instruction**.

Ensuite, notre processeur va passer par lire l'instruction dans le registre d'instruction et va en déduire comment configurer les circuits du processeur pour que ceux-ci exécutent l'instruction voulue : on dit que notre processeur va devoir décoder l'instruction. Une fois que c'est fait, le processeur va commander les circuits du processeur pour qu'ils exécutent l'instruction. Notre instruction va s'exécuter. C'est l'étape d'**exécution**. Cette exécution peut être un calcul, un échange de donnée avec la mémoire, des déplacements de données entre registres, ou un mélange des trois.

Ces deux étapes forment ce qu'on appelle l'***Instruction Cycle***. Tout processeur doit au minimum effectuer ces deux étapes dans l'ordre indiqué au dessus : *Fetch*, puis exécution. Il se peut que certains processeurs rajoutent une étape en plus, pour gérer certaines erreurs, mais on n'en parlera pas pour le moment.

Micro-instructions

Si tous les processeurs doivent gérer ces deux étapes, cela ne veut pas dire que chaque étape s'effectue d'un seul bloc. Chacune de ces étapes est elle-même découpée en plusieurs sous-étapes. Chacune de ces sous-étapes va aller échanger des données entre registres, effectuer un calcul, ou communiquer avec la mémoire. Pour l'étape de *Fetch* , on peut être sûr que tous les processeurs vont faire la même chose : il n'y a pas 36 façons pour lire une instruction depuis la mémoire. Mais cela change pour l'étape d'exécution : toutes les instructions n'ont pas les mêmes besoins suivant ce qu'elles font ou leur mode d'adressage. Voyons cela avec quelques exemples.

Instruction d'accès mémoires

Commençons par prendre l'exemple d'une instruction de lecture ou d'écriture. Pour commencer, on suppose que notre instruction utilise le mode d'adressage absolu. C'est à dire que cette instruction va indiquer l'adresse à laquelle lire dans sa suite de bits qui la représente en mémoire. L'exécution de notre instruction se fait donc en trois étapes : le chargement de l'instruction, le décodage, et la lecture proprement dite.

Étape 1	Étape 2
Fetch	Lecture de la donnée en mémoire

Mais si l'on utilise des modes d'adressages plus complexes, les choses changent un petit peu. Reprenons notre instruction Load, mais en utilisant une mode d'adressage utilisé pour des données plus complexe. Par exemple, on va prendre un mode d'adressage du style Base + Index. Avec ce mode d'adressage, l'adresse doit être calculée à partir d'une adresse de base, et d'un indice, les deux étant stockés dans des registres. En plus de devoir lire notre donnée, notre instruction va devoir calculer l'adresse en fonction du contenu fourni par deux registres.

Étape 1	Étape 3	Étape 4

Fetch	Calcul d'adresse	Lecture de la donnée en mémoire

Notre instruction s'effectue dorénavant en trois étapes, pas deux. Qui plus est, ces étapes sont assez différentes : une implique un calcul, et les autres impliquent un accès à la mémoire.

Instruction de calcul

Prenons maintenant le cas d'une instruction d'addition. Celle-ci va additionner deux opérandes, qui peuvent être soit des registres, soit des données placées en mémoires, soit des constantes. Si les deux opérandes sont dans un registre et que le résultat doit être placé dans un registre, alors la situation est assez simple. Il suffit simplement d'aller récupérer les opérandes dans les registres, effectuer notre calcul, et enregistrer le résultat. Suivant le processeur, cela peut se faire en une ou plusieurs étapes.

On peut ainsi avoir une seule étape qui effectue la récupération des opérandes dans les registres, le calcul, et l'enregistrement du résultat dans les registres.

Étape 1	Étape 2	Étape 3
Fetch	Decode	Récupération des opérandes, calcul, et enregistrement du résultat

Mais il se peut que sur certains processeurs, cela se passe en plusieurs étapes.

Étape 1	Étape 2	Étape 3	Étape 4	Étape 5
Fetch	Decode	Récupération des opérandes	Calcul	Enregistrement du résultat

Maintenant, autre exemple : une opérande est à aller chercher dans la mémoire, une autre dans un registre, et le résultat doit être enregistré dans un registre. On doit alors rajouter une étape : on doit aller chercher la donnée en mémoire.

Étape 1	Étape 2	Étape 3	Étape 4
Fetch	Decode	Lecture de la donnée en mémoire	Récupération des opérandes, calcul, et enregistrement du résultat

Et on peut aller plus loin en allant cherche notre première opérande en mémoire : il suffit d'utiliser le mode d'adressage *Base + Index* pour celle-ci. On doit alors rajouter une étape de calcul d'adresse en plus. Ne parlons pas des cas encore pire du style : une opérande en mémoire, l'autre dans un registre, et stocker le résultat en mémoire.

Étape 1	Étape 2	Étape 3	Étape 4	Étape 5
Fetch	Decode	Calcul d'adresse	Lecture de la donnée en mémoire	Récupération des opérandes, calcul, et enregistrement du résultat

Conclusion

Bref, on voit bien que l'exécution d'une instruction s'effectue en plusieurs sous-étapes bien distinctes. Chacune de ces étapes s'appelle une **micro-opération**. Chacune de ces micro-opération va : soit effectuer une lecture ou écriture en mémoire RAM, soit effectuer un calcul, soit échanger des données entre registres. **Chaque instruction machine est équivalente à une suite de micro-opérations exécutée dans un ordre précis**. C'est ainsi que fonctionne un processeur : chaque instruction machine est traduite en suite de micro-opérations lors de son exécution, à chaque fois qu'on l'exécute.

Pour créer un processeur qui fonctionne, on doit impérativement créer de quoi effectuer des micro-opérations, de quoi demander d'effectuer une micro-opération parmi toutes les autres, et de quoi les cadencer dans le bon ordre en fonction de l'instruction à exécuter. Pour cela, notre processeur va devoir être organisé d'une certaine façon, en plusieurs circuits.

L'intérieur d'un processeur

Dans les premiers ordinateurs, ces circuits étaient fusionnés en un seul gros circuit qui faisait tout. Pour se faciliter la tâche, les concepteurs de CPU ont décidé de segmenter les circuits du processeur en circuits spécialisés : des circuits chargés de faire des calculs, d'autres chargés de gérer les accès mémoires, etc.

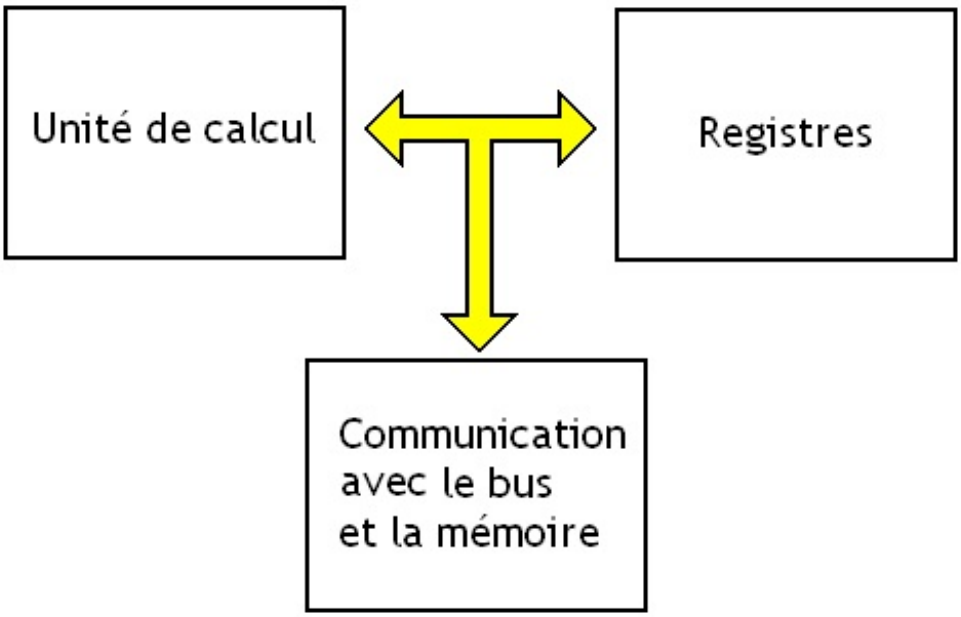
Le Datapath

Notre processeur dispose donc d'un ou de plusieurs circuits chargés d'effecteur des calculs : ce sont des **unités de calcul**. Ils servent pour effecteur des instructions de calcul, ou pour calculer des adresses mémoires (dans le mode d'adressage *Base + Index*, par exemple).

De même, notre processeur possède un circuit qui lui permet de communiquer avec la mémoire. notre processeur peut ainsi aller configurer des bus d'adresse, lire ou écrire sur le bus de donnée, etc. Sur les processeurs modernes, ce circuit s'appelle la ***Memory Management Unit*** .

Notre processeur dispose aussi de **registres**. Et il faut bien faire communiquer ces registres avec la mémoire, l'unité de calcul, ou faire communiquer ces registres entre eux ! Pour cela, notre processeur incorpore un bus permettant les échanges entre tout ce beau mode.

L'intérieur de notre processeur ressemble donc à ceci :



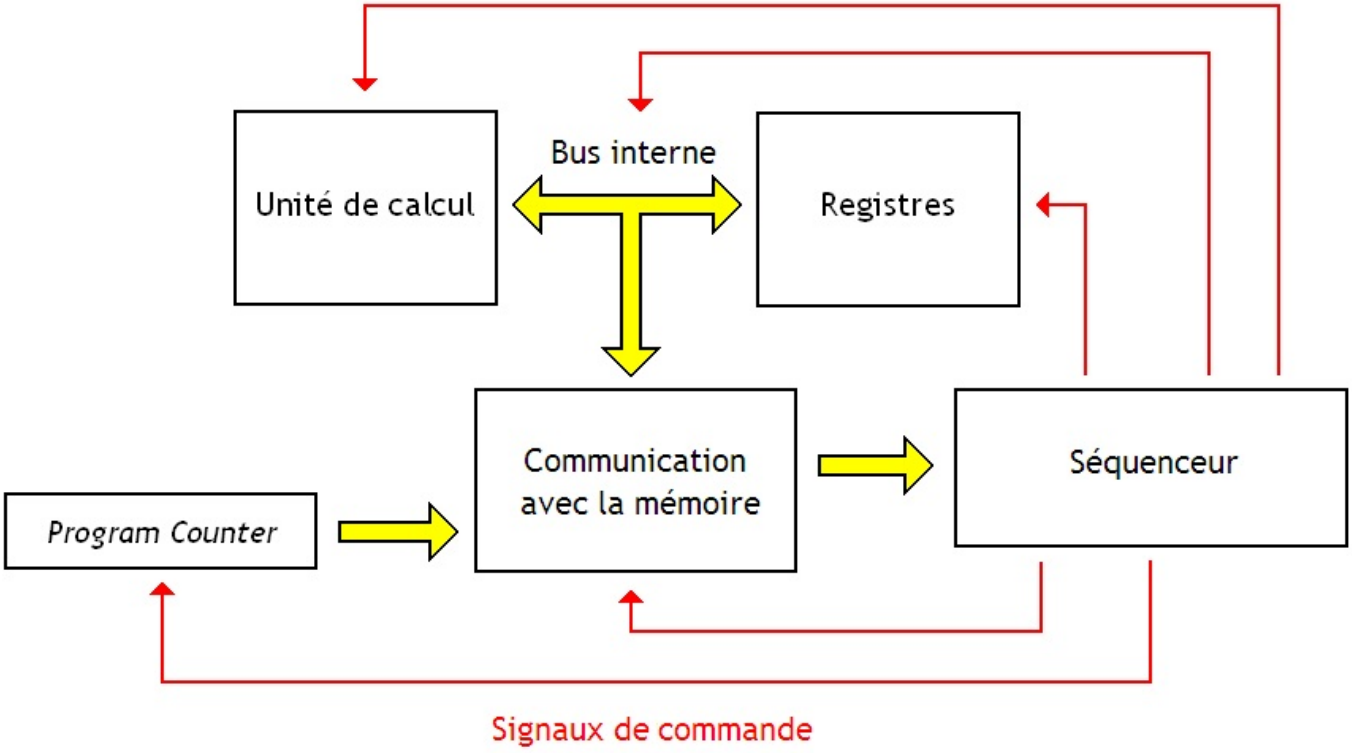
Cet ensemble minimal de composants nécessaire pour effectuer nos instructions s'appelle le ***Datapath***, ou chemin de données. On l'appelle ainsi parce qu'il regroupe tous les composants qui doivent manipuler des données.

Chaque micro-opération va effectuer une manipulation sur ce *Datapath*. Par exemple, une micro-opération pourra relier des registres sur les entrées de l'unité de calcul, demander à celle-ci de faire un calcul, et relier sa sortie sur un registre. Ou alors, une micro-opération pourra relier des registres sur le circuit permettant de communiquer avec le bus pour effecteur une lecture ou écriture. Ou alors, on pourra configurer notre bus de façon à copier le contenu d'un registre dans un autre. Bref, chacune des micro-opérations vue plus haut s'effectuera en configurant notre unité de calcul, en configurant notre bus, ou en configurant notre unité de communication avec la mémoire.

Le séquenceur

Mais pour pouvoir effectuer une instruction, il faut non seulement savoir effectuer chaque micro-opération de celle-ci, mais il faut aussi effectuer les bonnes micro-opérations dans le bon ordre. Par exemple, on ne doit pas actionner l'unité de calcul pour une lecture. Et il ne faut pas effectuer de calcul d'adresse pour une lecture en mode d'adressage absolu. Bref, il doit configurer le *Datapath*, et il doit effectuer les bonnes configurations les unes après les autres dans le bon ordre. Pour ce faire, notre processeur contient un circuit qui se charge d'effecteur chaque micro-opération nécessaire à l'exécution de notre instruction dans le bon ordre en fonction de l'instruction : c'est le **séquenceur**.

Ce séquenceur va envoyer des ordres aux autres circuits, afin de les configurer comme il faut. Ces ordres, ce sont des signaux de commande. L'intérieur de notre processeur ressemble donc à quelque chose dans le genre :



L'organisation illustrée dans le schéma est une organisation minimale, que tout processeur doit avoir. On peut la modifier et l'améliorer. On peut ainsi y rajouter des registres spécialisés comme un registre d'état, rajouter une connexion entre le *Program Counter* et le bus interne pour permettre les branchements relatifs ou indirects, rajouter des bus internes, etc. On peut aussi penser aux processeurs communiquant avec la mémoire par deux bus séparés : un pour une mémoire dédiée aux instructions, et un autre pour les données, etc. Bref, on ne va pas tout citer, il y en aurait pour une heure !

Maintenant qu'on sait quels sont les grands circuits présents dans notre processeur, on va voir ceux-ci plus en détail. Nous allons commencer par voir l'unité de calcul, et voir à quoi celle-ci ressemble de l'extérieur. Ensuite, nous verrons les registres du processeur, ainsi que l'interface avec la mémoire. Et enfin, nous regarderons l'organisation du bus interne, avant de passer au séquenceur, ainsi qu'à l'unité de *Fetch*. Un beau programme en perspective !

Les unités de calcul

Le rôle du chemin de donnée est d'exécuter une instruction et pour ce faire, notre processeur contient des circuits spécialement étudiés pour. Dans notre processeur, ces circuits sont presque toujours regroupés ensembles, avec quelques petits circuits supplémentaires pour gérer le tout, dans ce qu'on appelle une **unité de calcul**.

Pour simplifier, notre unité de calcul est donc une sorte de gros circuit qui contient de quoi faire des calculs arithmétiques ou des opérations logiques (comme des ET, OU, XOR, etc). Je ne vous cache pas que ces unités de calculs sont utilisés pour effectuer des instructions arithmétiques et logiques, pour les comparaisons et instructions de test. Mais on peut aussi l'utiliser lors des étapes de calcul d'adresse, de certains modes d'adressage. Après tout, calculer une adresse se fait avec des opérations arithmétiques et logiques simples, comme des décalages, des additions et des multiplications.

Cette unité de calcul est souvent appelée l'**unité arithmétique et logique** ou **UAL**. Certains (dont moi) préfèrent l'appellation anglaise *Arithmetic and logical unit*, ou **ALU**.

Une unité de calcul est capable d'effectuer des opérations arithmétiques, des opérations logiques, et des instructions de test (les

comparaisons).
Les instructions effectuées par ces unités de calcul sont donc le plus souvent :

- des additions ;
- des soustractions ;
- des multiplications ;
- des divisions ;
- des changements de signe ;
- des décalages logiques ;
- des décalages arithmétiques ;
- des rotations ;
- des NON logiques ;
- des ET logiques ;
- des OU logiques ;
- des XOR logiques ;
- des comparaisons ;
- des test d'égalité ;
- etc.

Certaines unités de calculs sont assez rudimentaires, et ne peuvent pas effectuer beaucoup d'instructions : on peut parfaitement créer des unités de calcul ne sachant faire que des opérations simples comme des ET, OU, et NON logiques. Par contre, d'autres unités de calcul sont plus évoluées et peuvent faire énormément d'instructions différentes : certaines unités de calcul sont capables d'effectuer toutes les instructions citées au-dessus, et quelques autres comme calculer des racines carrées, des tangentes, arc-tangentes, cosinus, etc.

Vu de l'extérieur

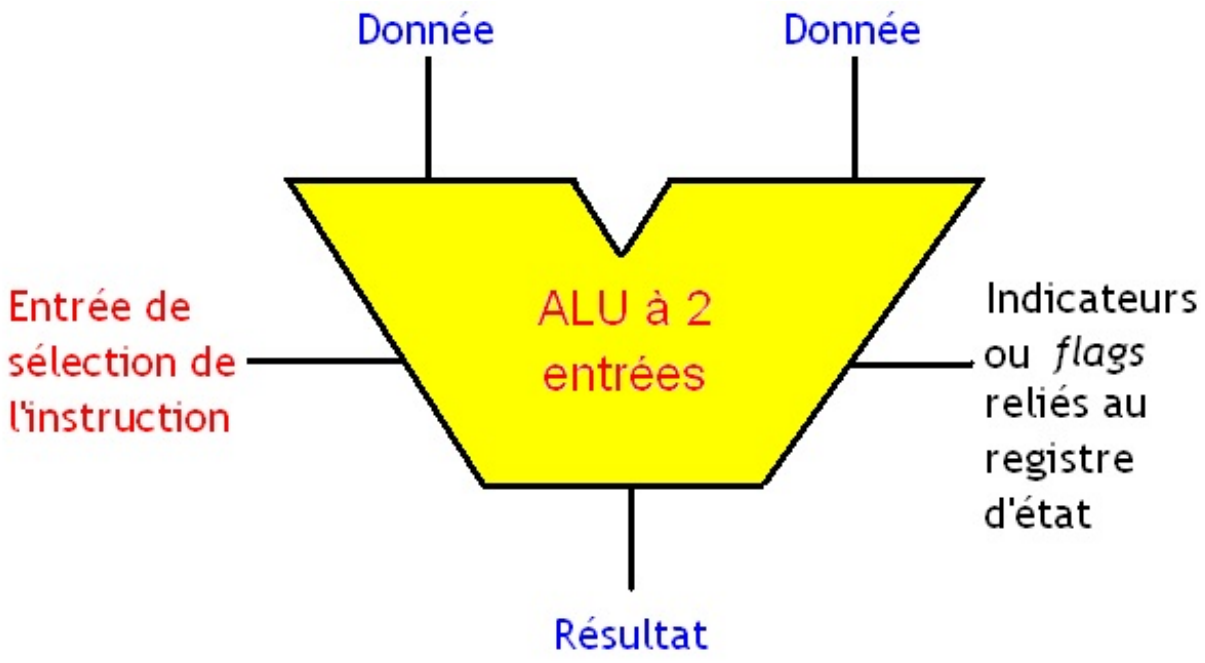
Nos instructions de comparaisons sont prises en charge par l'unité de calcul et elles doivent mettre à jour le registre d'état. Et c'est sans compter sur certaines opérations de calcul qui doivent mettre à jour le registre d'état : prenez l'exemple du bit *NULL*, cité il y a quelques chapitres. On peut donc en déduire que **le registre d'état est obligatoirement relié à certaines sorties de l'unité de calcul**.

De plus, il faudra bien spécifier l'instruction à effectuer à notre unité de calcul parmi toutes celles qu'elle est capable de faire. Après tout, il faut bien prévenir notre unité de calcul qu'on veut faire une addition et pas une multiplication : elle ne peut pas le deviner toute seule ! Il faut donc configurer notre unité de calcul pour que celle-ci exécute l'instruction voulue et pas une autre. Pour cela, notre unité de calcul possède une entrée permettant de la configurer convenablement. Cette entrée supplémentaire s'appelle l'**entrée de sélection de l'instruction**.

Sur cette entrée, on va mettre un nombre codé en binaire qui précise l'instruction à effectuer. Suivant le nombre qu'on va placer sur cette entrée de sélection de l'instruction, notre unité de calcul effectuera une addition, une multiplication un décalage, etc. Pour chaque unité de calcul, il existe donc une sorte de correspondance entre le nombre qu'on va placer sur l'entrée de sélection de l'instruction, et l'instruction à exécuter :

Entrée de sélection de l'instruction	Instruction effectuée par l'unité de calcul
0000	NOP
0111	Addition
1111	Multiplication
0100	ET logique
1100	Décalage à droite
...	...

Pour votre culture générale, voici comment sont généralement schématisées ces fameuses unités de calcul :



Ce schéma montre une unité de calcul effectuant des instructions sur une ou deux données à la fois : elle possède trois entrées et deux sorties.

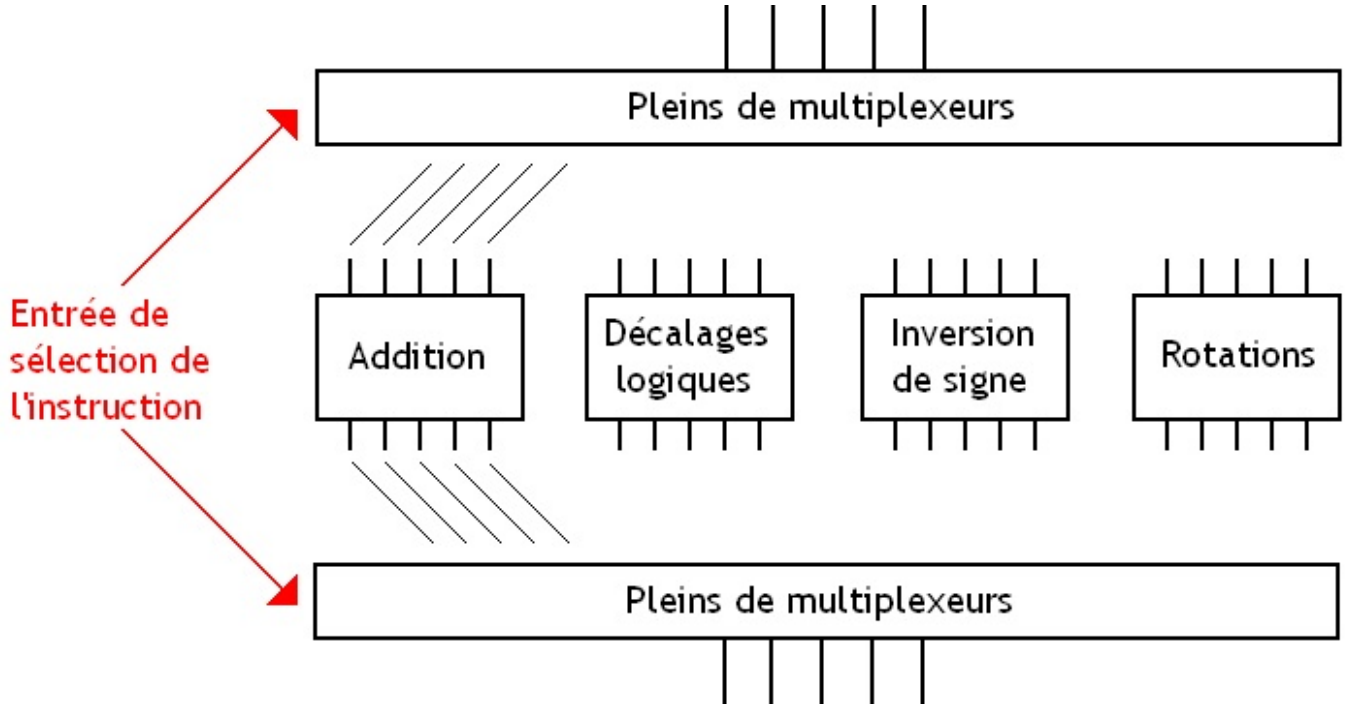
A l'intérieur d'une unité de calcul

Ces unités de calcul contiennent des circuits, fabriqués à base de portes logiques et de transistors. Suivant l'unité de calcul utilisée, son fonctionnement interne peut être plus ou moins compliqué.

Dans une ALU, il y a des circuits !

Certaines unités de calcul contiennent un circuit différent par instruction qu'elles sont capables d'exécuter. L'entrée de sélection sert donc uniquement à sélectionner le bon circuit et à connecter celui-ci aux entrées ainsi qu'aux sorties.

Pour sélectionner un de ces circuits, on utilise des circuits élèctroniques, qui vont se charger d'aiguiller les données placées en entrée sur le bon circuit. Ces circuits, qui vont relier le bon circuit aux entrées et aux sorties de notre unité de calcul, sont appelés des **multiplexeurs**. On peut voir ces multiplexeurs comme des espèces d'interrupteurs qui vont venir connecter ou déconnecter les entrées et sorties de notre unité de calcul au bon circuit.

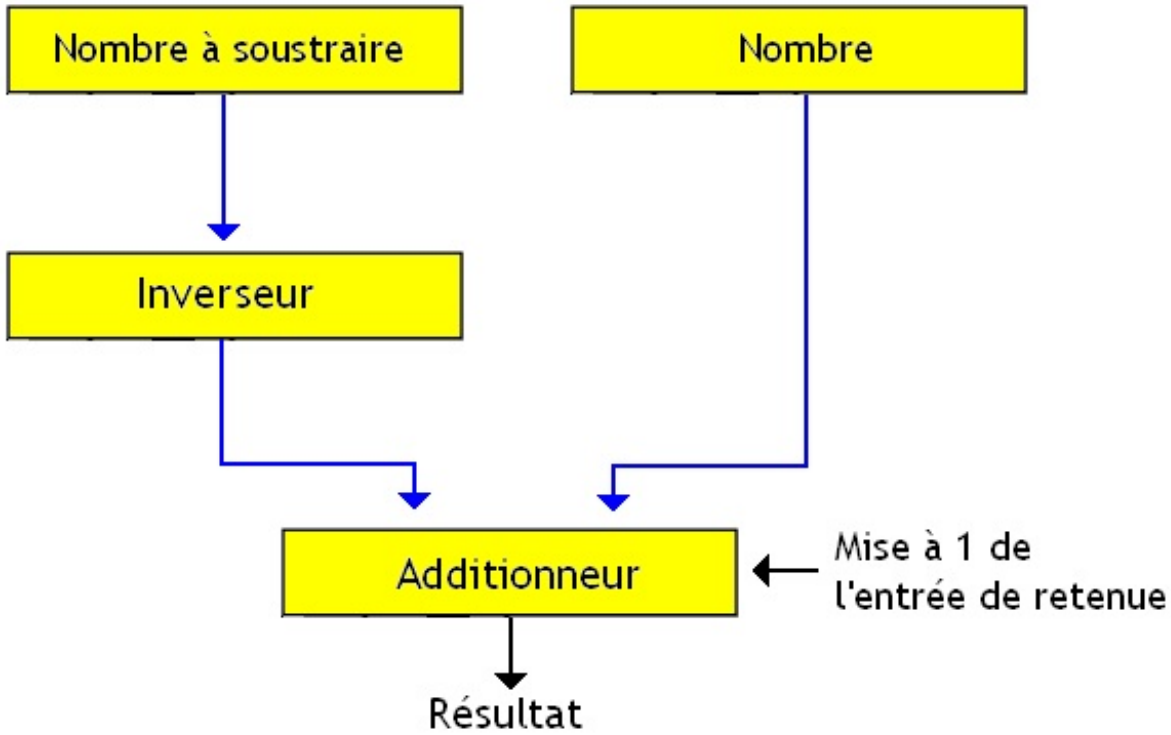


Bien sûr, ces multiplexeurs doivent être commandés pour se mettre dans la bonne position : l'entrée de sélection sert à cela.

Unité à configurer

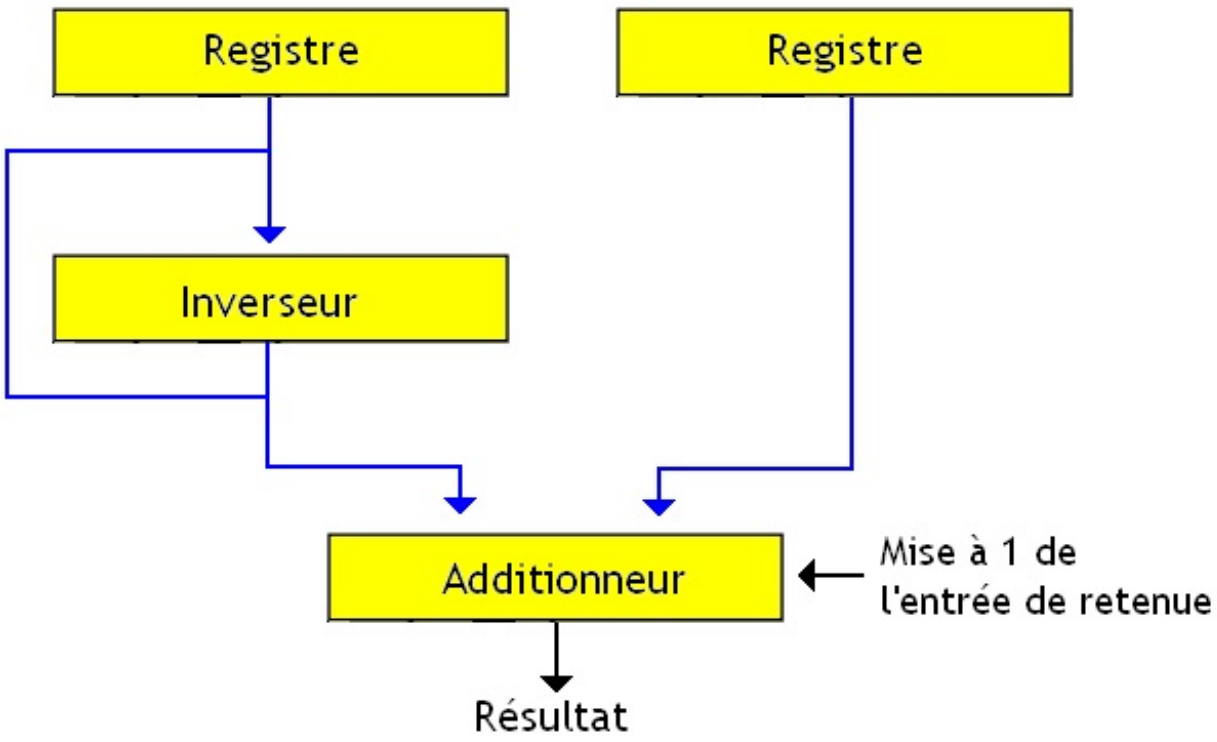
Utiliser un circuit pour chaque instruction semble être une solution assez sympathique. Mais si on regarde bien, elle possède un défaut assez important : certains morceaux de circuits sont présents en double dans notre unité de calcul. Cela ne se voit pas dans les schémas du haut, mais vous allez rapidement comprendre par un bref exemple.

Supposons que j'aie une ALU qui soit capable d'effectuer des additions et des soustractions. En utilisant la technique vue au-dessus, j'aurais donc besoin de deux circuits : un pour les additions et un pour les soustractions. Visiblement, il n'y a pas de duplications, à première vue. Mais si on regarde bien le circuit qui effectue la soustraction, on remarque que certains circuits sont présents en double. Notre soustracteur est composé de deux circuits, reliés en série. Il est composé d'un circuit qui inverse tous les bits de notre opérande, et d'un additionneur. On verra pourquoi au prochain chapitre.



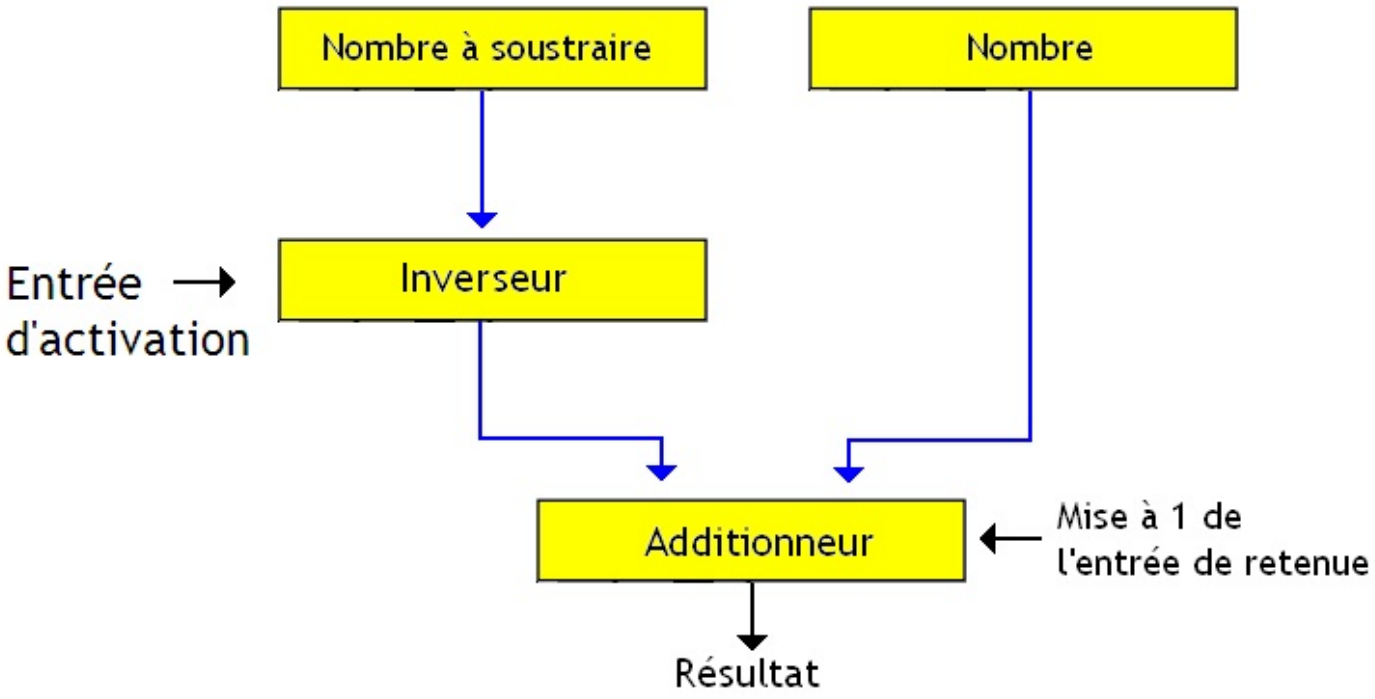
L'additionneur est donc présent en double : une fois dans le circuit chargé des soustractions et une autre fois dans celui capable d'effectuer les additions. Bien évidemment, il est plus économe en terme de circuits de ne pas avoir à placer deux additionneurs dans notre processeur. Pour éviter ce problème, on regroupe l'additionneur et l'inverseur dans un seul circuit et on sélectionne l'inverseur quand on veut effectuer une soustraction. Et ce qui peut être fait pour un additionneur peut aussi être fait pour d'autres circuits. Pour cela, certaines unités de calcul sont composées de circuits élémentaires qui sont sélectionnés ou dé-sélectionnés au besoin, suivant l'instruction à exécuter. Le choix des circuits à activer ou désactiver peut se faire de deux façons différentes.

Avec la première façon, ce choix est fait grâce à des "interrupteurs électroniques" (des transistors ou des multiplexeurs) qui connecteront entre eux les circuits à utiliser et déconnecteront les circuits qui sont inutiles pour effectuer l'instruction choisie. Suivant la valeur de l'entrée de sélection de l'instruction, quelques petits circuits internes à l'unité de calcul se chargeront de fermer ou d'ouvrir ces "interrupteurs" convenablement. Dans notre exemple, ce la reviendrait à faire ceci :



Pour spécifier qu'on veut faire une addition ou une soustraction, il faut alors placer une entrée chargée de spécifier s'il faut connecter ou déconnecter le circuit inverseur : cette entrée permettra de configurer le circuit. Ce qu'il faut mettre sur cette entrée sera déduit à partir de l'entrée de sélection d'instruction de l'ALU.

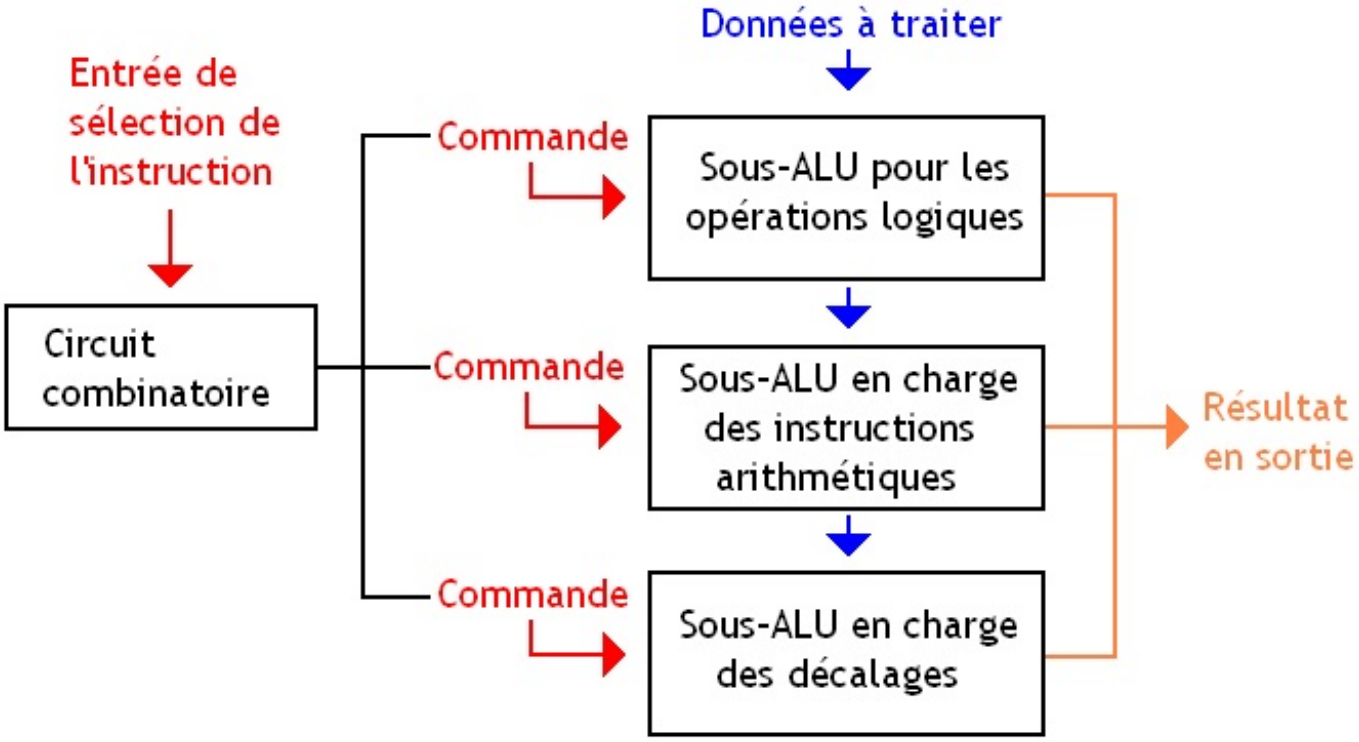
Une autre solution peut être de placer en série des circuits, qu'on va activer ou désactiver au besoin. Ces circuits possèdent une entrée qui leur demandera de fonctionner. Suivant la valeur de cette entrée, ces circuits vont agir différemment : soit ils vont faire ce qu'on leur demandent, soit ils recopieront leur entrée sur leur sortie (il ne feront rien, quoi).



Ce qu'il faut mettre sur cette entrée sera déduit à partir de l'entrée de sélection d'instruction de l'ALU.

Dans la réalité

les unités de calcul de nos processeurs actuels sont un mélange de tout ce qui a été vu au-dessus. Les unités de calcul les plus complètes sont en effet découpées en pleins de circuits, certains devant être configurés, d'autres reliés, etc. Tout ce passe comme si ces circuits se comportaient comme des sous-unités de calculs inclues dans une unité de calcul plus grosse. Par exemple, on peut avoir une sous-unité de calcul qui se charge des décalages et des rotations (un *barrel shifter*), une autre qui se charge des opérations logiques (ET, OU, NON, XOR), une autre pour les additions, soustractions et comparaisons, une autre pour la multiplication, etc.



Autant dire que l'intérieur d'une unité de calcul est assez sympathique. Dans ce genre de cas, une partie seulement de l'entrée de sélection sert à sélectionner un circuit, le reste servant à configurer le circuit sélectionné.

Bit Slicing

Sur certains processeurs assez anciens, l'ALU est elle-même découpée en plusieurs ALU plus petites, chacune capable d'effectuer toutes les instructions de l'ALU, reliées entre elles, qui traitent chacune un petit nombre de bits. Par exemple, l'ALU des processeurs AMD's Am2900 est une ALU de 16 bits composée de plusieurs sous-ALU de 4 bits. Cette technique qui consiste à créer des unités de calcul plus grosses à partir d'unités de calcul plus élémentaires s'appelle en jargon technique du **Bit Slicing**.

Unités annexes

Il y a au moins une unité de calcul dans un processeur. Ceci dit, il n'est pas rare qu'un processeur possède plusieurs unités de calcul. Cela peut avoir diverses raisons : cela peut-être pour des raisons de performance, pour simplifier la conception du processeur, ou pour autre chose.

FPU

Dans certains cas, le processeur disposent d'unités de calcul supplémentaires, capables d'effectuer des calculs sur des nombres flottants. Il faut dire que les circuits pour faire un calcul sur un entier ne sont pas les mêmes que les circuits faisant la même opération sur des flottants. Certains processeurs s'en moquent et utilisent une seule unité de calcul capable d'effectuer à la fois des calculs sur des entiers que des calculs sur des flottants.

Mais dans d'autres processeurs, les circuits calculant sur les entiers et ceux manipulant des flottants sont placées dans des unités de calculs séparées. On se retrouve donc avec une unité de calcul spécialisée dans le traitement des nombres flottants : cette unité s'appelle la ***Floating Point Unit***. On utilise souvent l'abréviation **FPU** pour désigner cette *Floating Point Unit*.

Les instructions supportées par les FPU et une unité de calcul sur des entiers sont souvent différentes. Les FPU peuvent effectuer d'autres instructions supplémentaires en plus des traditionnelles opérations arithmétiques : des racines carrées, des sinus, des cosinus, des logarithmes, etc. Et c'est normal : les résultats de ces opérations sont souvent des nombres à virgules et

rarement des nombres entiers.

ALU spécialisées

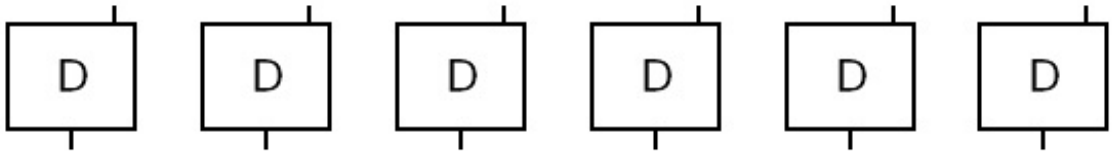
Un processeur peut aussi disposer d'unités de calcul spécialisées, séparées de l'unité de calcul principale. Par exemple, un processeur peut contenir des unités chargées d'effectuer des instructions de décalage, des unités spécialisées dans le calcul de certaines instructions, etc. Pour donner un exemple, les divisions sont parfois exécutées dans des circuits à part. Cette séparation est parfois nécessaire : certaines opérations sont assez compliquées à insérer dans une unité de calcul normale, et les garder à part peut simplifier la conception du processeur.

Registres et interface mémoire

Notre processeur contient un certain nombre de registres, nécessaires à son fonctionnement. La quantité de registres varie suivant le processeur, s'il s'agit d'un processeur RISC, CISC, ou de détails concernant son jeu d'instruction. Le nombre et le rôle des différents registres joue un rôle dans la façon dont on va concevoir notre processeur, et son *Datapath*.

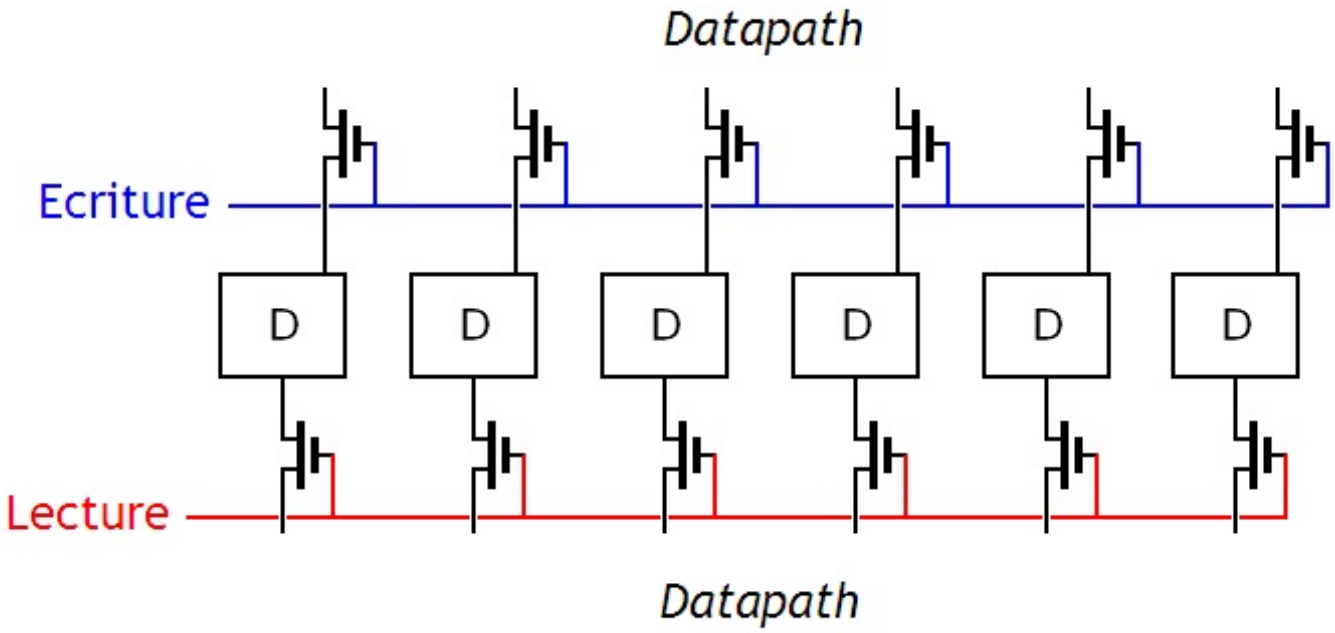
Registres simples

Comme on l'a vu il y a quelques chapitre, un registre est fabriqué à partir de mémoires de 1 bits qu'on appelle des bascules. Ces bascules possèdent une entrée sur laquelle on place un bit à mémoriser, et une sortie sur laquelle le bit mémorisé est disponible en permanence. En regroupant plusieurs de ces bascules ensembles, et en reliant les entrées d'autorisation d'écriture ensembles, on obtient un registre tout ce qu'il y a de plus simple.



Ce registre devra être lu et écrit, quel qu'il soit. Peut importe sa fonction dans le processeur, peu importe qu'il stocke une donnée, une adresse, une instruction, etc. Il devra pouvoir être écrit, lu, mais aussi être déconnecté du *Datapath* : il arrive qu'on ne veuille ni lire, ni écrire dans notre registre. Pour cela, on doit pouvoir connecter ou déconnecter notre registre au *Datapath*, suivant les besoins. De plus, on veut pouvoir spécifier si on effectue une lecture ou une écriture.

Pour cela, les entrées et les sorties de nos registres sont reliées au *Datapath* par des transistors. En ouvrant ou fermant ceux-ci, on peut déconnecter nos registres du *Datapath* , connecter les sorties des bascules sur le bus pour effectuer une lecture, ou connecter les sorties sur le bus pour une écriture.



On le voit sur ce schéma, certains transistors sont intercalés entre les sorties du registre et le reste du *Datapath* : ceux-ci servent à autoriser la lecture du registre. Les transistors intercalés entre les entrées et le reste du *Datapath* servent pour l'écriture.

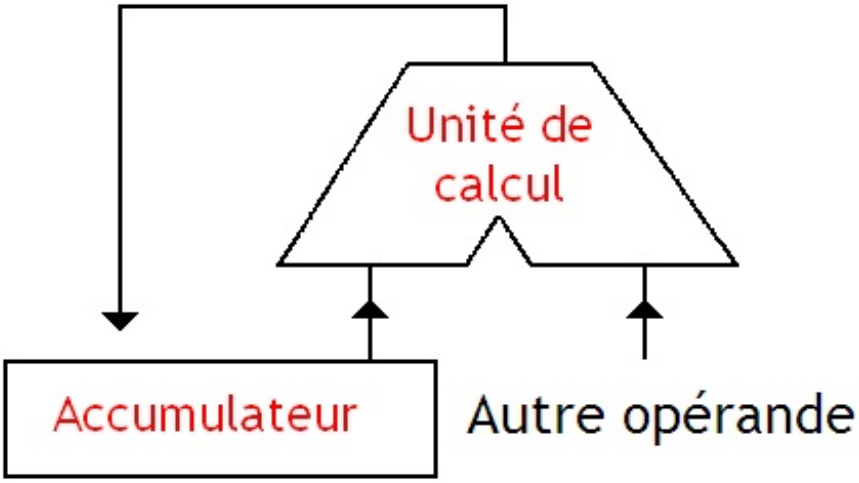
Chacun de ses ensemble de transistor va être regroupé au *Datapath* par un ensemble de fils qu'on appelle un port. Avec les registres de notre processeur, on dispose d'un au moins un port pour la lecture, et d'un port pour l'écriture pour chaque registre.

Tout les transistors d'un port sont commandés en même temps par un seul signal. Ce signal est généré par le séquenceur, en fonction de l'instruction à exécuter et du registre précisé par le mode d'adressage de l'instruction. Suivant la complexité du processeur, le nombre de registre, et l'architecture, générer les signaux pour chaque registre peut être plus ou moins complexe.

Registres non-référencables

De nombreux registres d'un processeur n'ont pas de noms de registres ou d'adresse. C'est le cas du *Program Counter* sur la majorité des processeurs. Idem pour le registre d'état, ou certains autres registres. Ceux-ci sont obligatoirement sélectionnés implicitement par certaines instructions, seules autorisées à en modifier le contenu. Ces registres sont simplement implémentés comme indiqué au-dessus, directement avec des bascules. Ils sont reliés au *Datapath* d'une façon ou d'une autre, et le séquenceur se charge de fournir les signaux de lecture et d'écriture aux transistors.

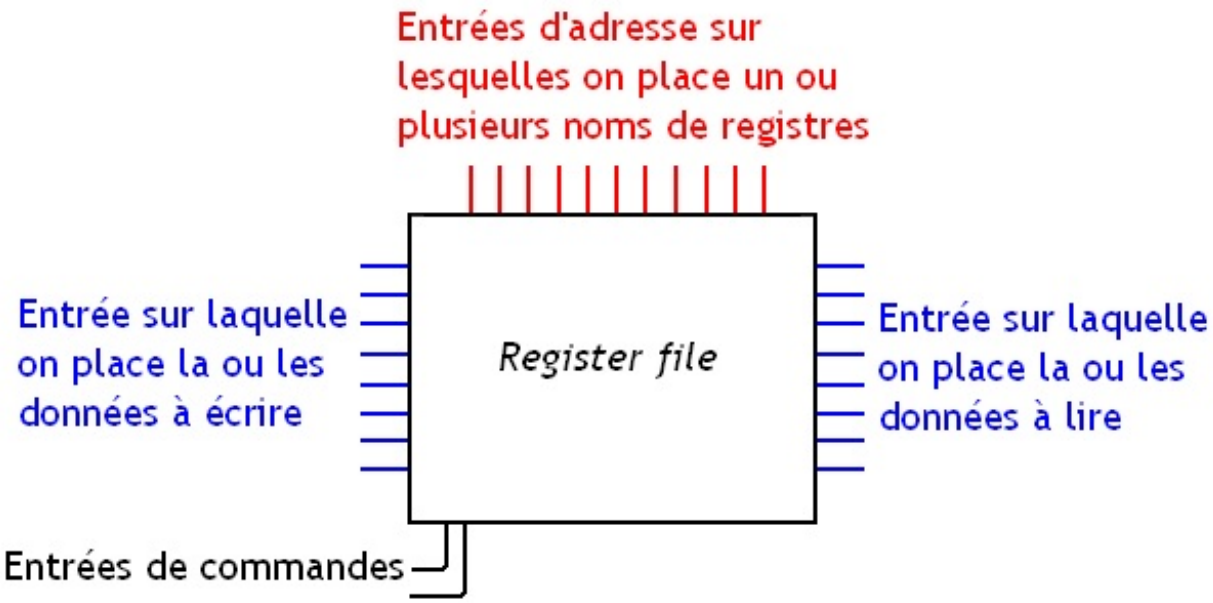
On peut citer le cas du registre accumulateur, présent sur les architectures à accumulateur. Cet accumulateur est implémenté le plus simplement du monde : il s'agit d'un regroupement de bascules, sans rien d'autre. A chaque fois qu'une instruction doit utiliser l'accumulateur, le séquenceur va autoriser les écritures dans celui-ci, en mettant à 1 le signal d'écriture. Celui-ci est reliée à l'unité de calcul de cette façon :



Register File

Mais un processeur se contente rarement d'un simple accumulateur. Dans la plupart des cas, notre processeur incorpore des tas de registres plus ou moins variés. Dans ce cas, on devra pouvoir sélectionner les registres à manipuler parmi tous les autres. Cela se fait en utilisant des noms de registres (ou éventuellement des adresses, mais passons).

Sur les processeurs utilisant des noms de registres, on rassemble nos registres dans un seul grand composant qu'on appelle le ***Register File***. On peut voir ce *Register File* comme une sorte de grosse mémoire dont chaque case mémoire serait un registre. Il s'agit réellement d'une mémoire : ce *Register File* contient des entrées qui permettent de sélectionner un registre, des entrées de commande, et des entrées sur lesquelles on envoie des données à lire ou écrire.



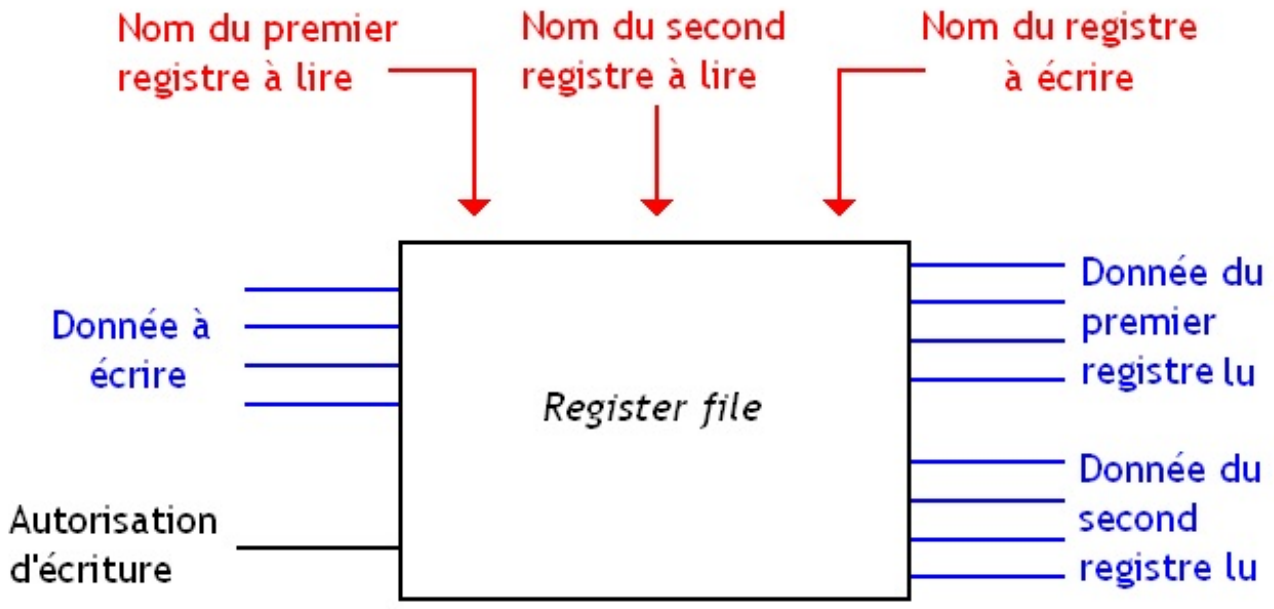
Comme ce schéma l'indique, ce *Register File* contient une entrée d'adresse sur laquelle on place une suite de bit qui permet d'identifier le registre à sélectionner. Et oui, vous avez devinés : cette suite de bit n'est autre que le nom du registre en question. Sur les processeurs avec un *Register File*, on peut voir le nom d'un registre comme une sorte d'adresse permettant d'identifier un registre dans le *Register File*.

Bien sûr, il y a des exceptions. Il ne faut pas oublier que certains registres n'ont pas de noms : le *Program Counter*, le registre d'état, etc. Ceux-ci ne sont pas forcément rassemblés avec les autres registres et sont souvent intégrés dans des circuits spécialisés ou mis à part des autres registres. Ce n'est toutefois pas systématique : on peut placer ces registres dans un *Register File*, mais c'est rarement utilisé. Dans ce cas, on doit jouer un peu sur les noms de registre avant de les envoyer sur les entrées d'adresse du *Register File*. Rien de bien méchant.

Mémoire multiports

Comme vous l'avez remarqué sur le schéma, le *Register File* a une petite particularité : les écritures et les lectures ne passent pas par les mêmes ensembles de fils. On a un ensemble de fils pour la lecture, et un autre pour les écritures. Tout se passe comme si notre *Register File* était relié à deux bus de données : un pour les lectures, et un pour les écritures. Ces ensembles de fils s'appellent des **ports**. Vu que notre *Register File* possède plusieurs de ces ports, on dit que c'est une **mémoire multiports**.

Mais cette technique, qui consiste à fournir plusieurs ensembles de fils pour des lectures ou écriture va souvent plus loin : on ne se limite pas à deux ports : on peut en avoir bien plus ! On peut avoir plusieurs ports pour les lectures, voire plusieurs ports pour les écritures. Cela permet d'effectuer plusieurs lectures ou écritures simultanément.

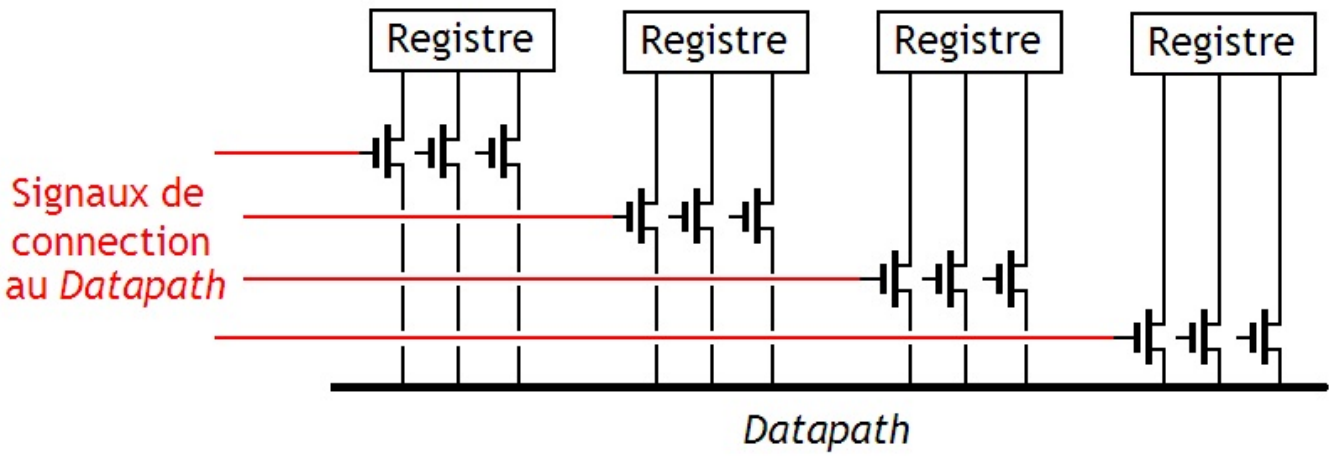


Cela permet de lire ou d'écrire dans plusieurs registres en une seule fois si le reste du processeur est adapté. Mine de rien, beaucoup d'instructions ont besoin de lire plusieurs registres à la fois : une addition a besoin de deux opérandes, pareil pour la soustraction, la comparaison, la multiplications, etc. Beaucoup d'instructions de nos processeurs ont besoin de lire deux données pour donner leur résultat : elles sont dites dyadiques. Et ne parlons pas des (rares) instructions ayant besoin de trois opérandes. Autant dire que créer des mémoires permettant de lire plusieurs registres à la fois sont un plus appréciable.

Implémentation

Ce *Register File* est un circuit qui est tout de même assez facile à fabriquer. Dans celui-ci, on trouve des registres, des transistors, et un nouveau composant qu'on appelle un décodeur. Voyons tout cela en détail. L'idée de base dérriére notre *Register File* est de rassembler plusieurs registres ensembles, et de les connecter au même *Datapath* via leurs ports. On se retrouve avec nos registres, ainsi que les transistors reliés à chaque port. Pour éviter les conflits, un seul registre par port devra être sélectionné. Pour cela, on devra commander correctement les transistors qui relient les registres au *Datapath*.

Ici, exemple avec un port de lecture

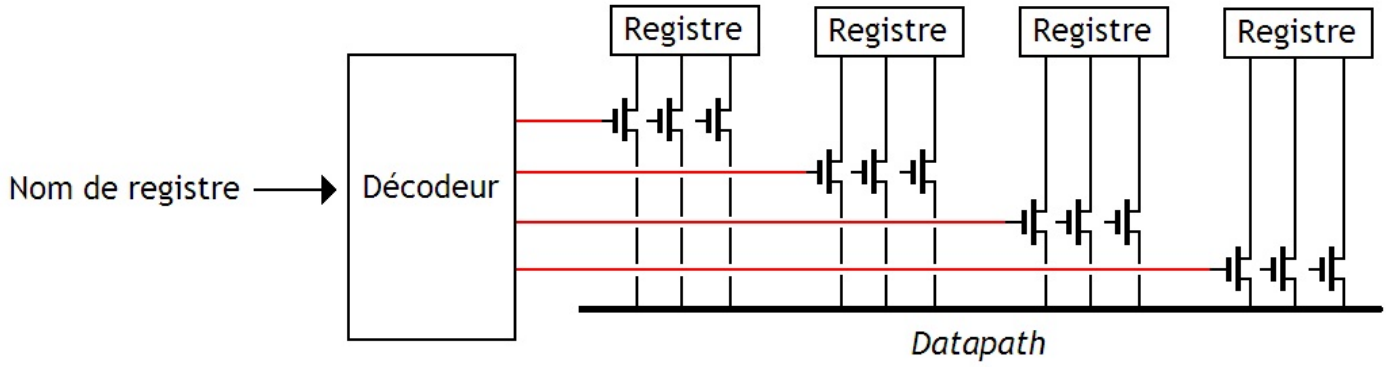


En sélectionnant un registre par son nom, on doit activer un seul de ces signaux de commande à la fois. On a donc une correspondance entre un nom de registre, et un de ces signal. Cette correspondance est unique : à un signal, un nom de registre, et réciproquement. La traduction entre nom de registre et signaux de commande est effectuée par un circuit spécialisé. Celui-ci doit répondre à plusieurs exigences :

- Il doit partir d'un nom de registre codé sur n bits : ce circuit a donc n entrées ;

- notre nom de registre de n bits peut adresser 2^n registres différents : notre circuit doit donc posséder 2^n sorties ;
- chacune de ces sorties sera reliée à un signal de commande et permettra de connecter ou déconnecter un registre sur le port ;
- on ne doit sélectionner qu'un registre à la fois : une seule sortie devra être placée à 1, et toutes les autres à zéro ;
- et enfin, deux noms de registres différents devront sélectionner des registres différents : la sortie de notre circuit qui sera mise à 1 sera différente pour deux noms de registres différents.

Il existe un composant électronique qui répond à ce cahier des charges : le **décodeur**.



Ce décodeur est, comme tous les autres circuits électroniques, conçu avec des portes logiques. Dans sa version la plus naïve, on peut créer un décodeur en utilisant les techniques vues au chapitre 3 : on établit une table de vérité, qu'on transforme en équations logiques, et on traduit le tout en circuit. Vous êtes donc censé savoir le faire. J'ai donc l'honneur de vous annoncer que vous savez comment implémenter un port d'un *Register File* : il suffit de prendre des registres, des transistors, un décodeur, et roulez jeunesse !

Pour un *Register File* utilisant plusieurs ports, on devra utiliser plusieurs décodeurs, et multiplier les transistors reliés sur la sortie ou l'entrée de chaque registre. Il faut savoir que plus un *Register File* a de ports, plus il utilisera de circuits : les décodeurs et transistors supplémentaires ne sont pas gratuits. En conséquence, il aura tendance à consommer du courant et à chauffer. La quantité d'énergie consommée par une mémoire est proportionnelle à son nombre de ports : plus on en met, plus notre mémoire va consommer de courant et chauffer. Et notre *Register File* ne fait pas exception à la règle. Les concepteurs de processeurs sont donc obligés de faire des compromis entre le nombre de ports du *Register File* (et donc la performance du processeur), et la chaleur dégagée par le processeur.

Register Files séparés

Utiliser un gros *Register File* n'est pas sans défauts. Un gros *Register File* consomme beaucoup d'énergie et chauffe beaucoup. Sans compter qu'un gros *Register File* est plus lent qu'un petit. Quand le nombre de registres augmente dans le processeur, il devient difficile d'utiliser un seul gros *Register File*. On doit alors trouver une autre solution. Et celle-ci est assez simple : il suffit de scinder ce gros *Register File* en plusieurs *Register Files* séparés. Utiliser plusieurs *Register Files* séparés a des avantages : cela consomme nettement moins d'énergie, rend le *Register File* plus rapide, et chauffe beaucoup moins. Cela rend le processeur légèrement plus difficile à fabriquer, mais le jeu en vaut la chandelle.

C'est assez souvent utilisé sur les architectures dont les registres sont spécialisés, et ne peuvent contenir qu'un type bien défini de donnée. Sur ces processeurs, certains registres sont conçus pour stocker uniquement des entiers, des flottants, des adresses, etc. On peut ainsi avoir un *Register File* pour les registres chargés de stocker des flottants, un autre pour les registres stockant des entiers, un autre pour les adresses, etc.

Mais rien ne nous empêche d'utiliser plusieurs *Register Files* sur un processeur qui utilise des registres généraux, non spécialisés. Cela s'utilise quand le nombre de registres du processeur devient vraiment important. Il faut juste prévoir de quoi échanger des données entre les différents *Register Files*.

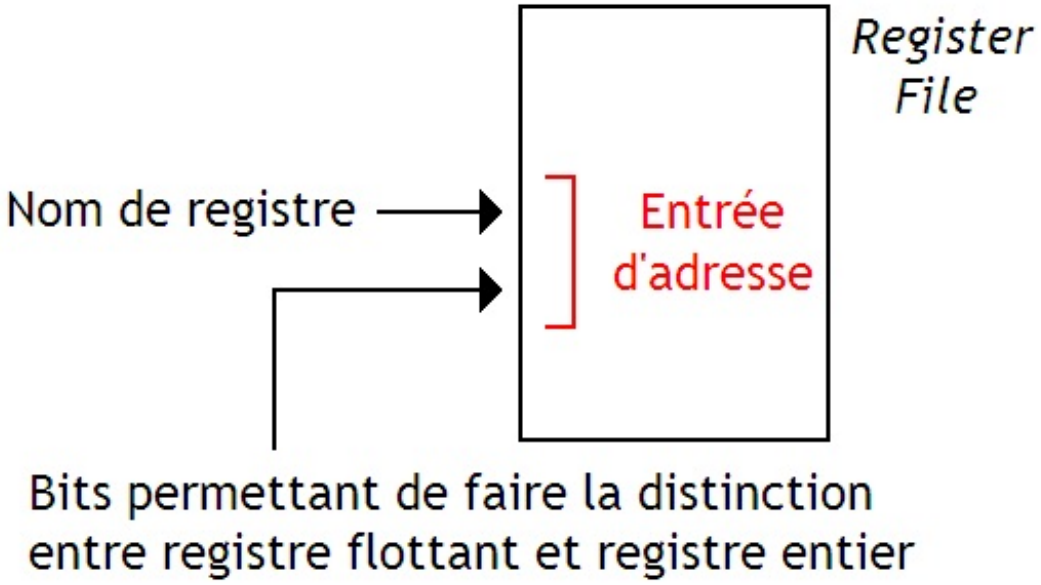
Fichier de registre Unifié

Séparer des registres spécialisés dans des *Register Files* différents n'est pas une obligation : rien ne nous empêche de regrouper tout ces registres spécialisés dans un seul gros *Register File* qui rassemble presque tous les registres.

Unification flottants/entiers

Par exemple, c'est le cas des Pentium Pro, Pentium II, Pentium III, ou des Pentium M : ces processeurs ont des registres séparés

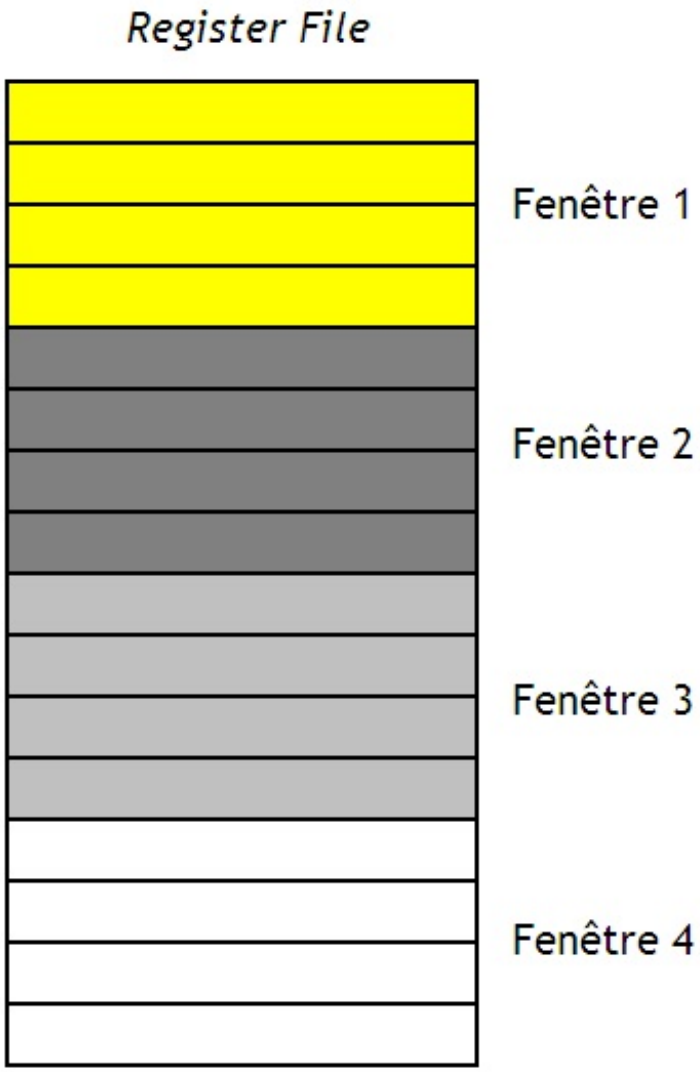
pour les flottants et les entiers, mais ils sont regroupés dans un seul *Register File*. Avec cette organisation, un registre flottant et un registre entier peuvent avoir le même nom de registre. Et il faudra faire la distinction. Pour cela, des bits vont être ajoutés au nom de registre. L'ensemble formera un nouveau nom de registre, qui sera envoyé sur un port du *Register File*. Ces bits supplémentaires sont fournis par le séquenceur, qui déduit leur valeur en fonction de l'instruction qu'il a décodé.



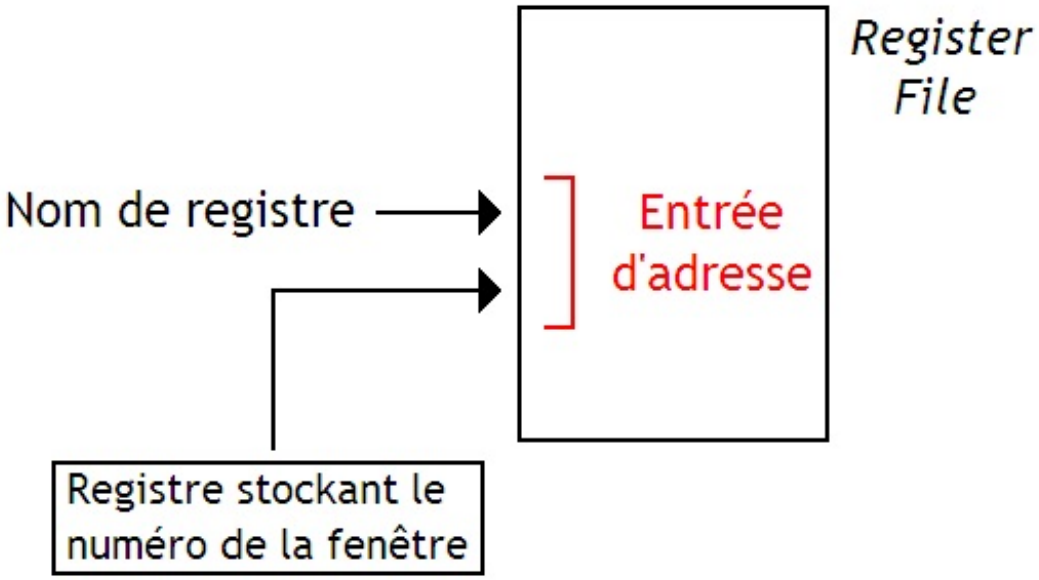
Sur certains processeurs, cette technique va encore plus loin : le Program Counter et/ou le registres d'état sont placés directement dans le *Register File*, avec éventuellement d'autres registres spécialisés, comme le *Stack Pointer*. Il va de soit que ce genre de technique est tout de même relativement maléfique.

Register Windowing

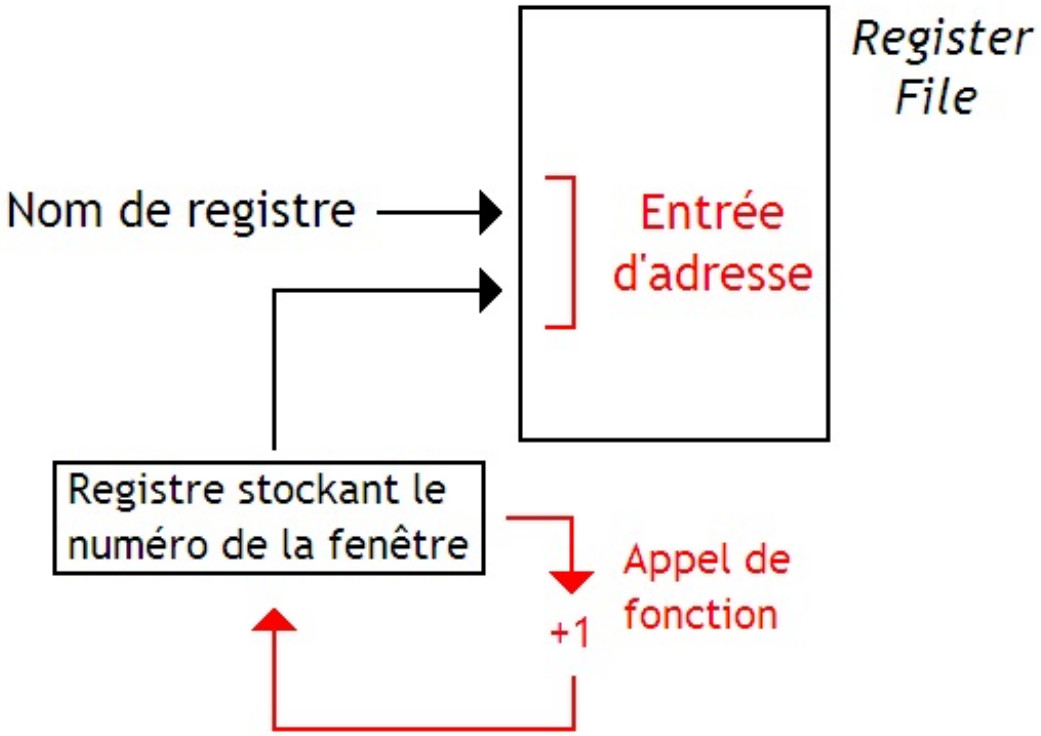
L'utilisation d'un *Register File* qui regroupe tous les registres du processeur permet d'implémenter facilement le fenêtrage de registres. Il suffit pour cela de regrouper tous les registres des différentes fenêtres dans un seul *Register File*.



Avec l'organisation indiquée dans ce schéma, le numéro d'un registre, tel qu'il est précisé par une instruction, n'est pas le nom de registre que l'on va envoyer sur le *Register File*. On devra rajouter quelques bits pour faire la différence entre les fenêtres. Cela implique de se souvenir dans quelle fenêtre de registre on est actuellement. Pour cela, toutes nos fenêtres sont numérotées. Avec ca, il ne nous reste plus qu'à ajouter un registre dans lequel on stocke le numéro de la fenêtre courante. Le contenu de ce registre sera concaténé au numéro du registre à accéder, afin de forer le numéro de registre à envoyer sur le *Register File*.



Pour changer de fenêtre, on doit modifier le contenu de ce registre. Cela peut se faire assez simplement : il suffit de passer à la fenêtre suivante ou précédente à chaque appel ou retour de fonction. En considérant que la fenêtre initiale est la fenêtre 0, la suivante sera 1, puis la 2, puis la 3, etc. Bref, il suffit d'ajouter 1 pour passer à la fenêtre suivante, et retrancher 1 pour passer à la fenêtre précédente. Cela se fait grâce à un petit circuit combinatoire relié à ce registre, qui est activé par le séquenceur à chaque appel/retour de fonction.



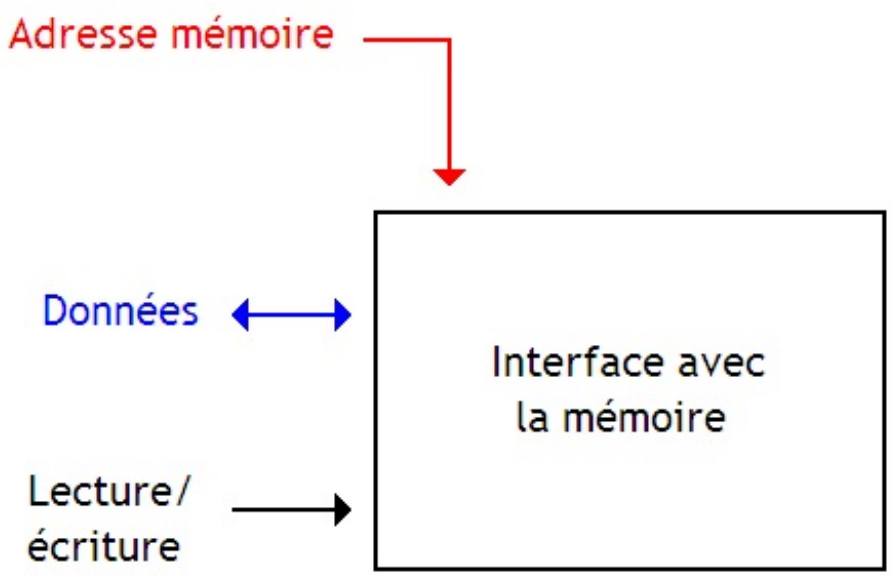
Bien sûr, il faut aussi prendre en compte le cas où ce registre déborde : si toutes les fenêtres sont occupées, il faut bien faire quelque chose ! Cela nécessite de rajouter des circuits qui s'occuperont de gérer la situation.

Communication avec la mémoire

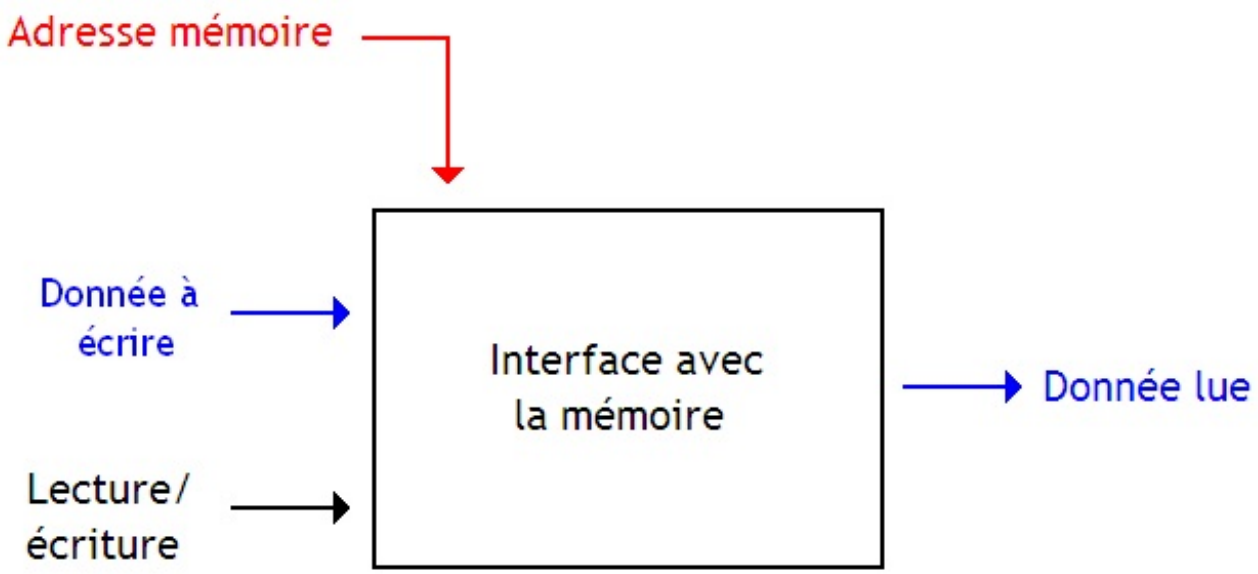
La communication avec la mémoire est assez simple : il suffit juste de placer l'adresse à laquelle accéder sur le bus d'adresse, configurer le fameux bit R/W qui permet de dire si on veut lire ou écrire, et éventuellement écrire une donnée sur le bus de donnée dans le cas d'une écriture. Rien de plus simple, donc.

Unité de communication avec la mémoire

Pour cela, notre unité de communication avec la mémoire doit donc avoir deux entrées : une pour spécifier l'adresse à laquelle lire ou écrire, et une qui permettra d'indiquer si on veut effectuer une lecture ou une écriture. Elle doit aussi avoir au moins une entrée/sortie connectée au bus de donnée.



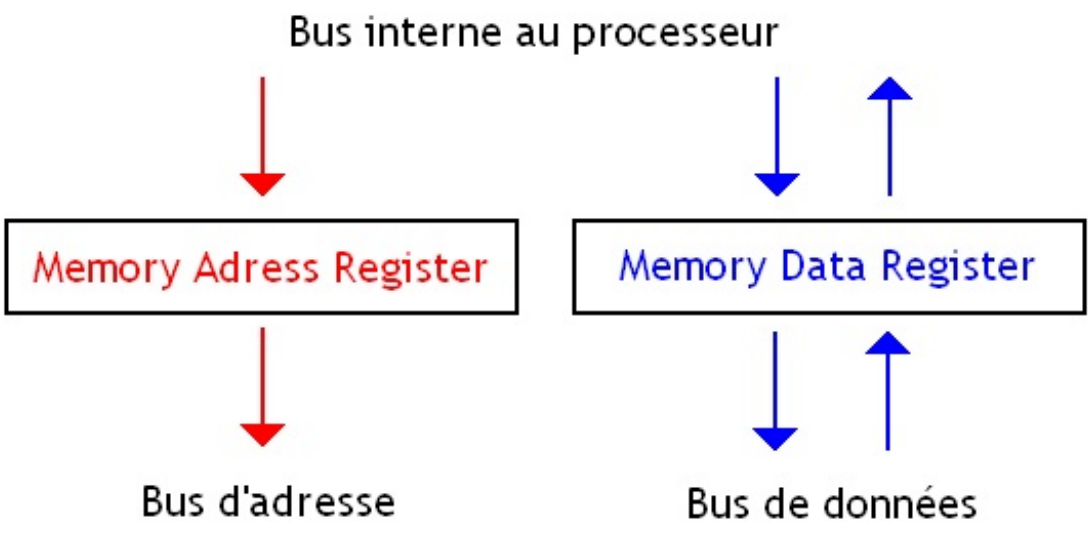
Si je dis au moins une, c'est parce qu'il est possible que la lecture et l'écriture ne se fassent pas par les mêmes ports.



Ces ports peuvent être reliés directement aux bus d'adresse, de commande ou de données. On peut aussi insérer des multiplexeurs entre les ports de notre unité de communication et les bus, histoire d'aiguiller les informations du bon port vers le bon bus. Bref.

Registres d'interfaçage mémoire

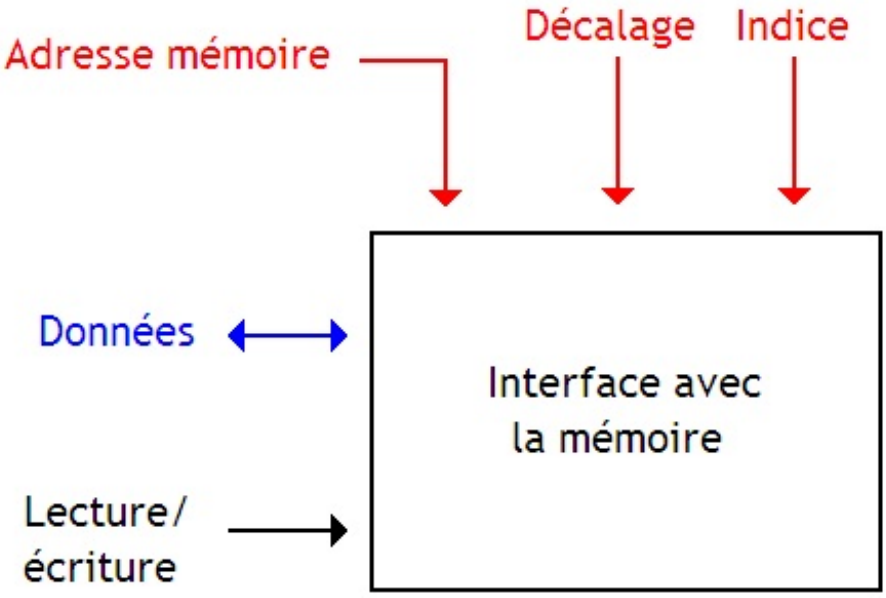
Pour se faciliter la vie, et pour simplifier la conception du processeur, ces ports sont parfois reliées à des registres : les registres d'interfaçage mémoire. Ainsi, au lieu d'aller lire ou écrire des données et des adresses directement sur le bus, on peut alors simplement aller lire ou écrire dans des registres spécialisés. Les registres d'interfaçage mémoire sont intercalés entre le bus mémoire et le reste du processeur (son chemin de données). Généralement, on trouve deux registres d'interfaçage mémoire : un registre d'adresse, relié au bus d'adresse ; et un registre de donnée, relié au bus de donnée.



Voilà à quoi ressemble notre unité de communication avec la mémoire. Certaines unités sont un peu plus intelligentes et sont capables de faire les calculs d'adresses nécessaires pour certains modes d'adressages. Au lieu d'utiliser notre ALU pour faire ces calculs, on peut la libérer pour en faire autre chose. Mais cela nécessite de dupliquer des circuits, ce qui n'est pas gratuit.

AGU

Enfin, il faut savoir que certaines unité de communication avec la mémoire ne prennent pas qu'une adresse en entrée. Elles peuvent gérer directement certains modes d'adressages elle-mêmes, et la calculer à partir d'une adresse, d'un indice, et éventuellement d'un décalage.



Ce calcul se fait alors soit dans l'unité de communication avec la mémoire, soit directement lors de l'accès mémoire lui-même. Dans le cas où c'est l'unité de communication avec la mémoire qui s'occupe de faire ce calcul d'adresse, celle-ci contient une petite unité de calcul interne, l'*Adress Generation Unit*. Dans l'autre cas, le calcul d'adresse s'effectue directement lors de l'accès mémoire : la mémoire fusionne les circuits de calcul d'adresse avec le décodeur.

Le chemin de données

Pour échanger des informations entre les composants du *Datapath*, on utilise un ou plusieurs **bus internes au processeur** qui relie tous les registres et l'unité de calcul entre eux. Bien évidemment, il faudra bien gérer ce bus correctement pour pouvoir échanger des informations entre unités de calculs et registres. Ce serait dommage de se tromper de registre et d'écrire au mauvais endroit, ou de choisir la mauvaise instruction à exécuter. Cette gestion du bus interne sera déléguée au séquenceur (qui ne fait pas que çà, bien sûr).

Une histoire de connexion

Imaginons que je souhaite exécuter une instruction qui recopie le contenu du registre R0 dans R2 : il va me falloir sélectionner les deux registres correctement via ce bus. De même, si je veux exécuter une instruction comme une racine carrée sur le contenu d'un registre, je dois fatalement connecter le bon registre sur l'entrée de l'ALU. Bref, quelque soit la situation, je dois relier certains composants sur le bus correctement, et déconnecter tous les autres.

Pour cela, chaque composant d'un processeur est relié au bus via des interrupteurs. Il nous suffit d'intercaler un interrupteur sur chaque fil qui sépare le bus de notre composant. Pour sélectionner le bon registre ou l'unité de calcul, il suffira de fermer les bons interrupteurs et d'ouvrir les autres. Le principe de gestion du chemin de donnée reste le même : **pour exécuter une étape d'une instruction, il faut commander des interrupteurs correctement**, et éventuellement configurer l'ALU et le *register file* pour choisir la bonne instruction et les bons registres. Tout cela est géré par le séquenceur.

On remarque immédiatement une chose : chaque interrupteur doit être commandé : on doit lui dire s'il doit se fermer ou s'ouvrir. Oui, vous avez vus où je veux en venir : cet interrupteur n'est rien d'autre qu'un transistor (on utilise aussi des multiplexeurs dans certaines situations).

Mais comment on fait pour préciser le sens de transfert des données sur le bus ?

En fait, on a pas vraiment besoin de préciser le sens de transfert. L'écriture et la lecture ne se font pas par les mêmes broches, quelque soit le composant à sélectionner : cela et vrai pour les ALU ou pour les registres. On a un ensemble d'entrées pour la lecture, et un ensemble de sorties spécialement dédiées pour l'écriture. Ces ensembles d'entrée-sortie forment ce qu'on appelle un **port** d'entrée ou de sortie. Tout ce qu'il faut faire, c'est ouvrir ou fermer les interrupteurs reliant le bon port au bus interne.

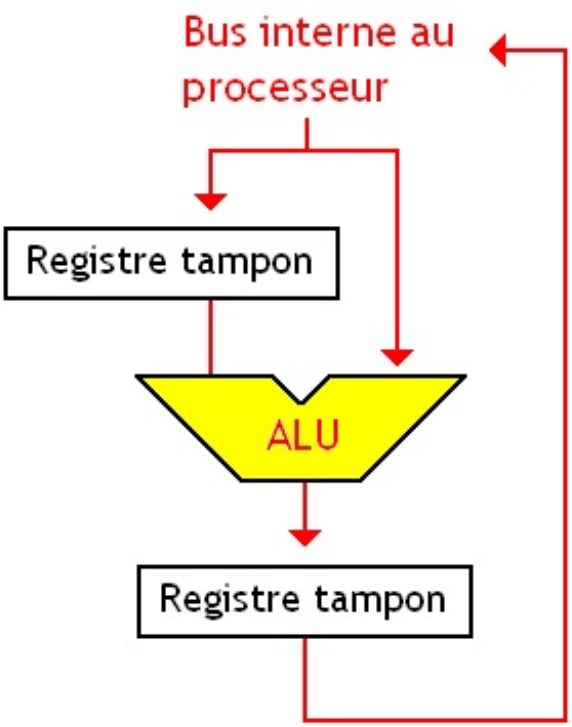
Chemin de donnée à un seul bus

Dans le cas le plus simple, nos processeurs utilisent un seul bus interne. Sur ces processeurs, l'exécution d'une instruction ne travaillant que dans des registres pouvait prendre plusieurs étapes.

Étape 1	Étape 2	Étape 3	Étape 4
Fetch	Récupération des opérandes	Calcul	Enregistrement du résultat

Cela arrive quand notre instruction doit effectuer un calcul nécessitant plusieurs opérandes.

Prenons un exemple : imaginez qu'on souhaite effectuer une addition entre deux registres. Il faut deux nombres pour que notre addition fonctionne. Il va nous falloir relier l'ALU à ces deux registres, ce qui impossible avec un seul bus : on ne peut envoyer qu'une seule donnée à la fois sur le bus ! On pourra donc avoir accès à une donnée, mais pas à l'autre. Pour résoudre ce problème, on a pas vraiment le choix : on doit utiliser un registre temporaire, directement relié à notre ALU, qui stockera une des deux donnée nécessaire à notre instruction. De même, il est préférable d'utiliser un registre temporaire pour stocker le résultat : comme cela, on évite que celui-ci se retrouve immédiatement sur le bus de donnée en même temps que la seconde opérande.



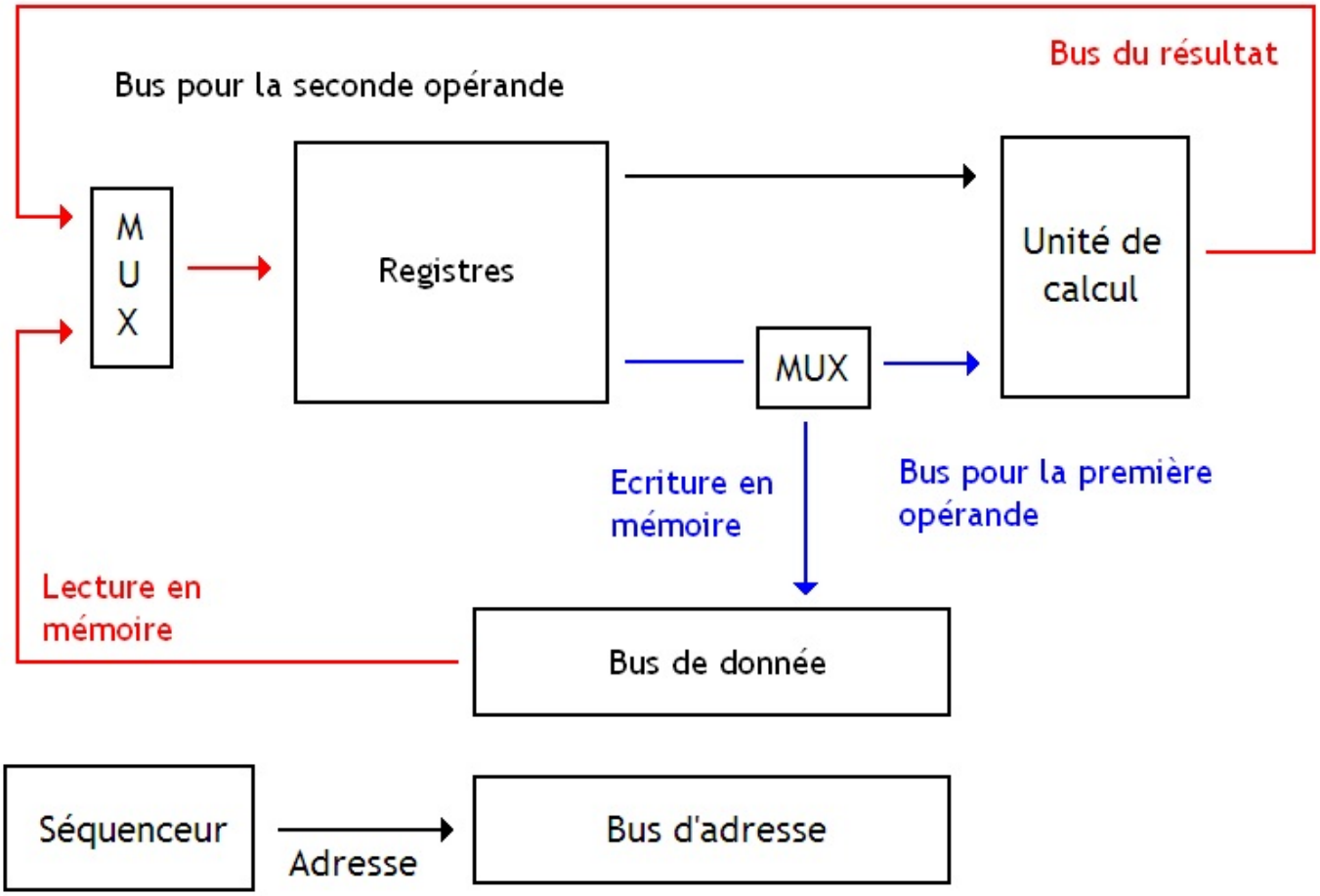
Le déroulement d'une addition est donc simple : il faut recopier la première donnée dans le registre temporaire, connecter le registre contenant la deuxième donnée sur l'entrée de l'unité de calcul, lancer l'adddition en envoyant le bon code sur l'entrée de sélection de l'instruction de l'ALU, et recopier le résultat dans le bon registre. Cela se fait donc en plusieurs étapes, chacune d'entre elle devant configurer le chemin de données.

Et avec plusieurs bus ?

Certains processeurs s'arrangent pour relier les composants du Datapath en utilisant plusieurs bus, et en utilisant un *register file* multiport. Cela permet de simplifier la conception du processeur ou d'améliorer ses performances. Par exemple, en utilisant plusieurs bus internes, un calcul dont les opérandes sont dans des registres et dont le résultat être stocké dans un registre peut se faire en une seule étape (souvenez-vous du début de ce chapitre). On gagne donc en rapidité.

Étape 1	Étape 2
Fetch	Récupération des opérandes, calcul, et enregistrement du résultat

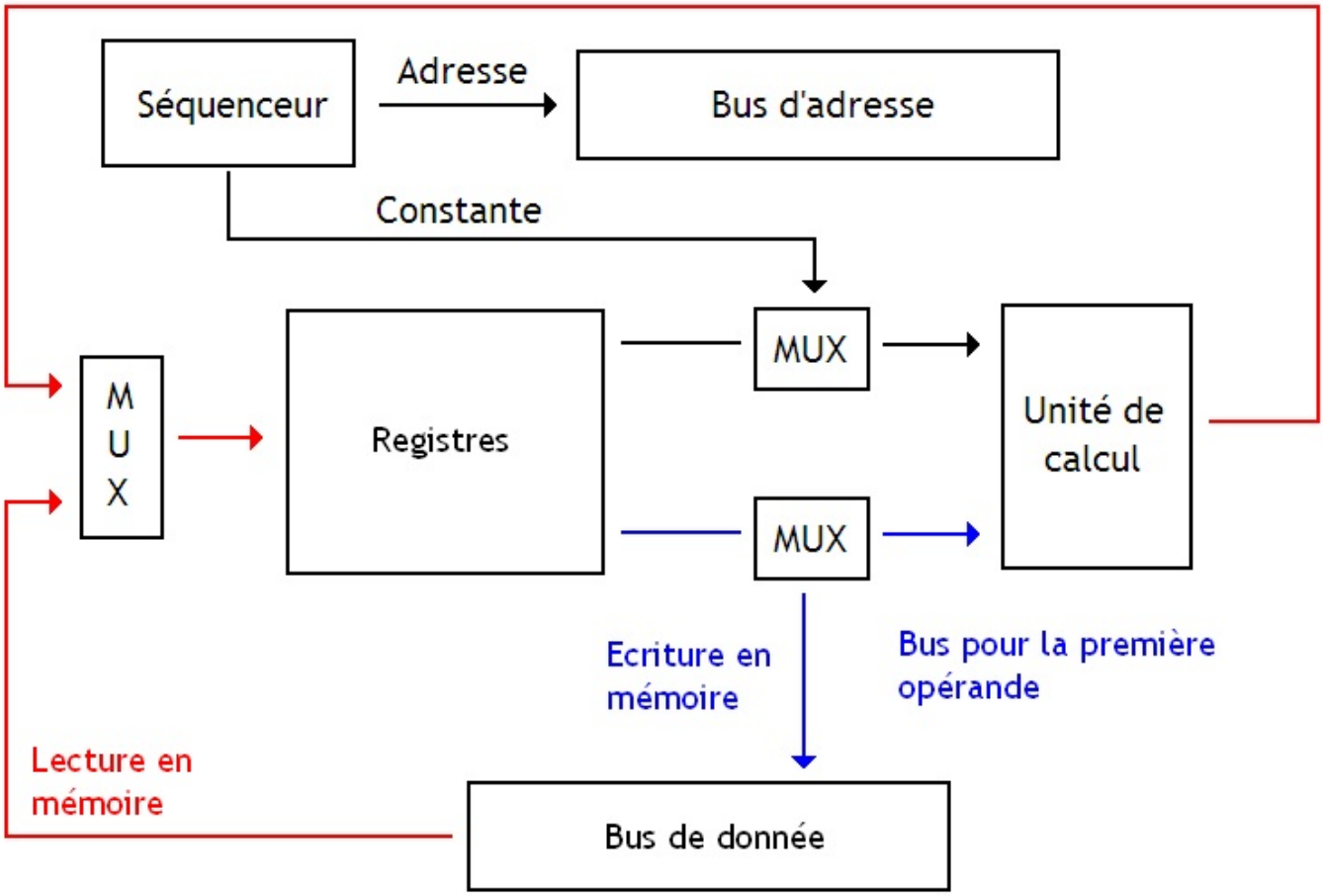
Pour cela, il suffit d'utiliser trois bus, reliés sur nos registres, l'ALU et le bus de données comme indiqué dans le schéma qui suit.



Avec cette organisation, notre processeur peut gérer les modes d'adressage absolus, et à registre, pas plus. C'est en quelque sorte l'architecture minimale de tout processeur utilisant des registres. Avec une organisation plus complexe, on peut gérer d'autres modes d'adressage plus facilement.

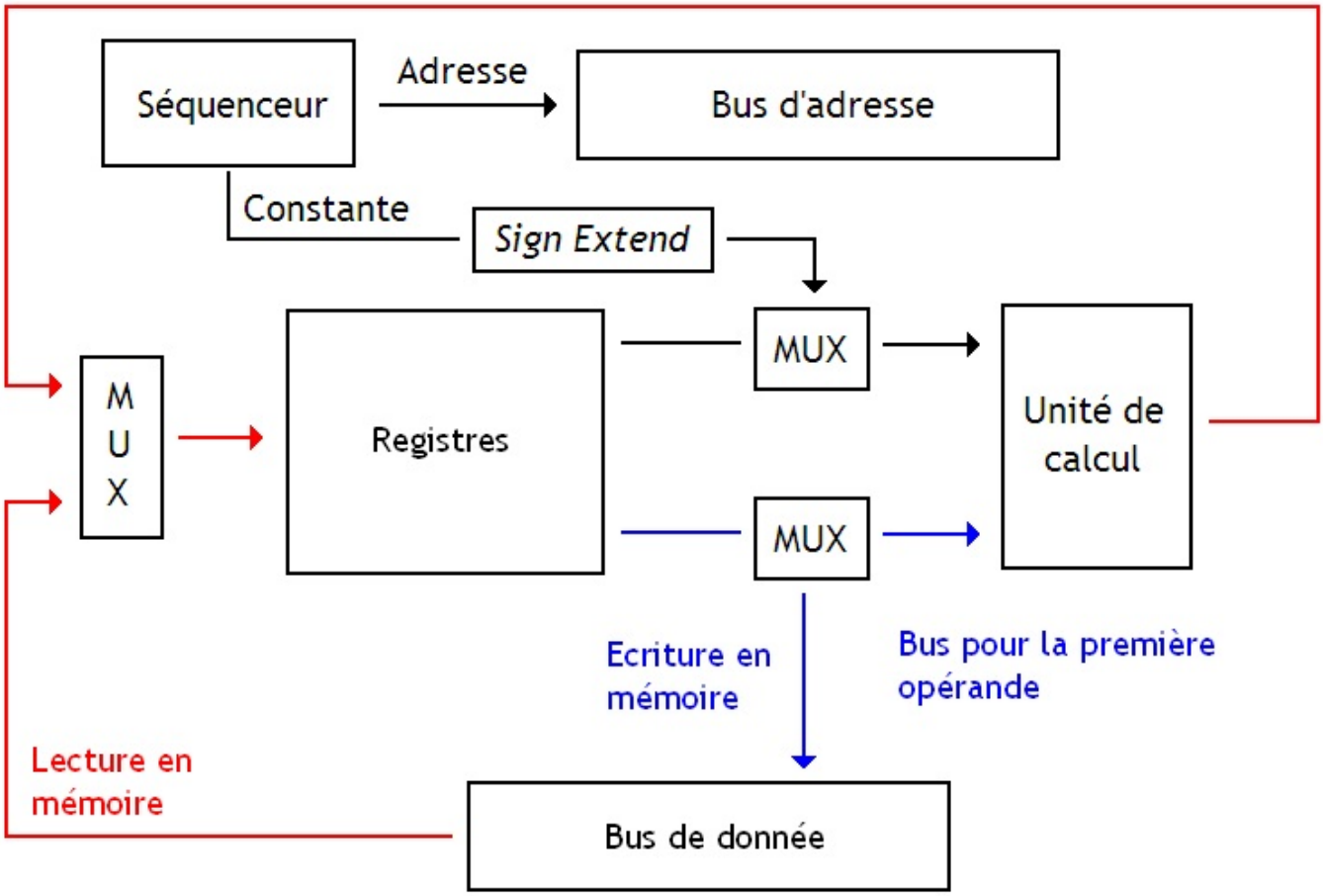
Adressage immédiat

Pour gérer l'adressage immédiat, on n'a pas vraiment le choix : on doit placer la constante inscrite dans l'instruction sur l'entrée de notre ALU. Cette constante est fournie par le séquenceur : lorsque celui-ci décode l'instruction, il va savoir qu'un morceau de l'instruction est une constante, et il va pouvoir l'envoyer directement sur l'ALU. Pour ce faire, une sortie du séquenceur va être relié sur l'entrée de notre ALU via un des bus. Seul problème : on ne va pas pouvoir rajouter un bus exprès pour le séquenceur. Généralement, on réutilise un bus qui sert pour relier les registres à l'entrée de l'ALU, et on lui permet d'être connecté soit sur cette sortie du séquenceur, soit sur nos registres. On utilise pour cela un gros paquet de multiplexeurs.



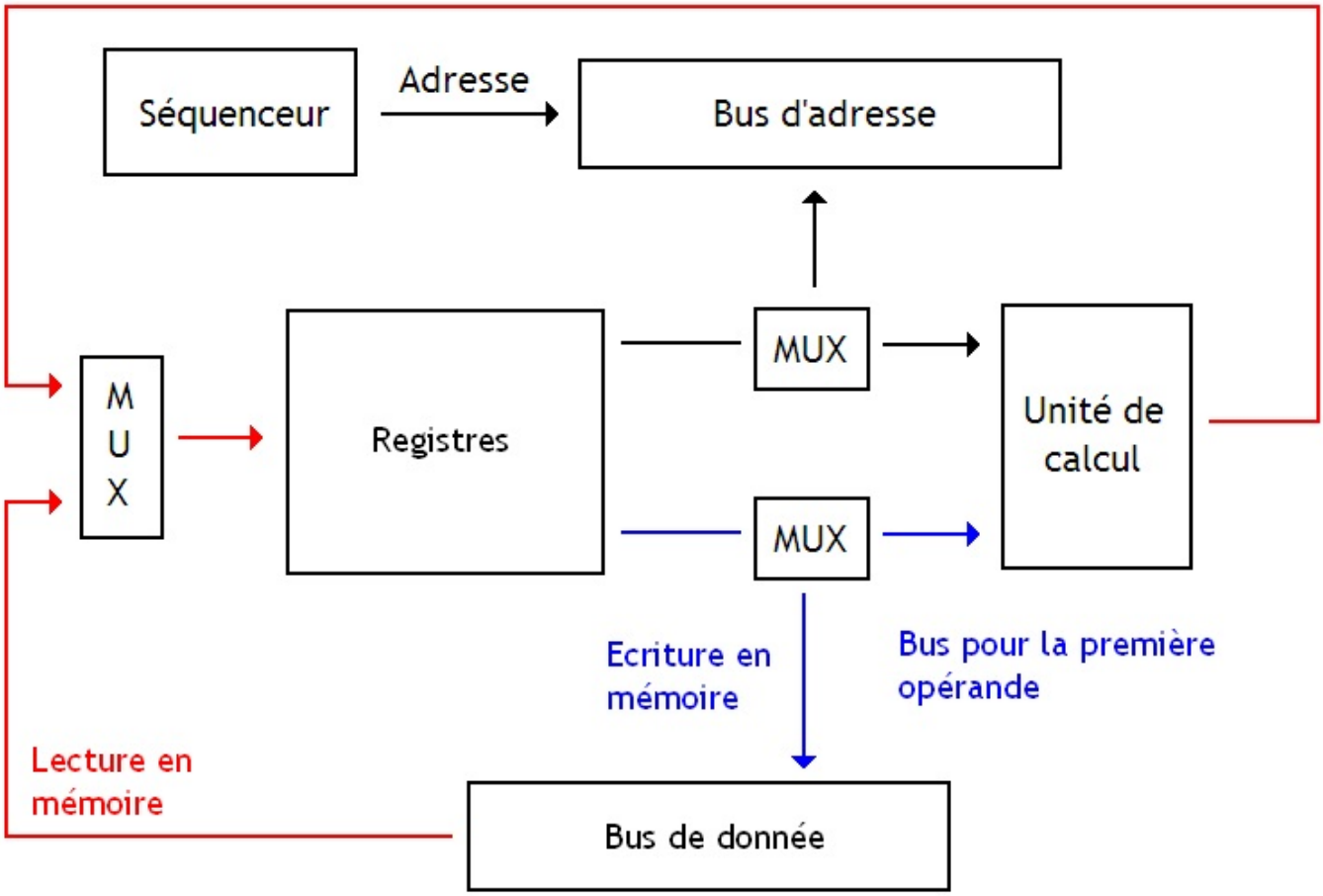
Sur certains processeurs, cette constante n'est pas toujours positive, et peut aussi être négative. Elle est généralement codée en complément à deux, et ne prend pas autant de bits que ce qu'un registre permet de stocker. Ces constantes sont souvent codées sur 8 ou 16 bits : aller au delà serait inutile vu que la quasi-totalité des constantes manipulées par des opérations arithmétiques sont très petites et tiennent dans un ou deux octets. Le seul problème, c'est que pour être envoyée en entrée de l'unité de calcul, une constante doit être convertie en un nombre de même taille que ce que peut manipuler notre ALU. Par exemple, si notre ALU manipule des données sur 32 bits et que nos registres font 32 bits, une constante codée sur 8 ou 16 bits devra être convertie en un nombre de 32 bits. Cela se fait en recopiant le bit de signe dans les bits de poids fort supplémentaires (rappelez-vous, on a vu ça dans le premier chapitre).

Pour effectuer cette extension de signe, on peut soit implanter un circuit spécialisé qui s'occupera d'effectuer cette extension de signe directement, soit on utilise l'ALU pour effectuer cette extension de signe, on enregistre notre constante convertie dans un registre, et on effectue notre calcul.



Adressage indirect à registre

La gestion du mode d'adressage indirect à registre est assez simple. Il suffit juste de relier un des bus interne du processeur sur le bus d'adresse, ou sur le registre d'interfaçage mémoire adéquat. Il suffit alors de relier le bon registre (celui qui contient l'adresse à laquelle accéder) sur le bus d'adresse. Si on veut effectuer une écriture, il suffit d'envoyer la donnée à écrire sur le bus de donnée via un autre bus interne au processeur.



Conclusion

Bref, je suppose que vous voyez le principe : on peut toujours adapter l'organisation des bus internes de notre processeur pour gérer de nouveaux modes d'adressages, ou pour améliorer la gestion des modes d'adressages existants. On peut ainsi diminuer le nombre d'étapes nécessaires pour éxecuter une instruction en ajoutant des bus ou en modifiant leur configuration.

Par exemple, on peut faire en sorte que les instructions manipulant une seule de leurs opérandes en mémoire et qui stockent leur résultat dans un registre s'effectuent en une seule étape : il suffit d'adapter l'organisation vue au-dessus en reliant le bus de donnée sur l'entrée de l'unité de calcul. On peut aussi gérer des instructions manipulant toutes leurs opérandes en mémoire en utilisant plusieurs registres d'interfaçage, qui peuvent être reliés aux entrées des unités de calcul, etc. Bref, l'organisation du chemin de donnée d'un processeur dépend fortement de ses modes d'adressage, et de sa conception : deux processeurs avec des modes d'adressage identiques seront conçus différemment, et n'effectueront pas forcément les même étapes pour éxecuter la même instruction.

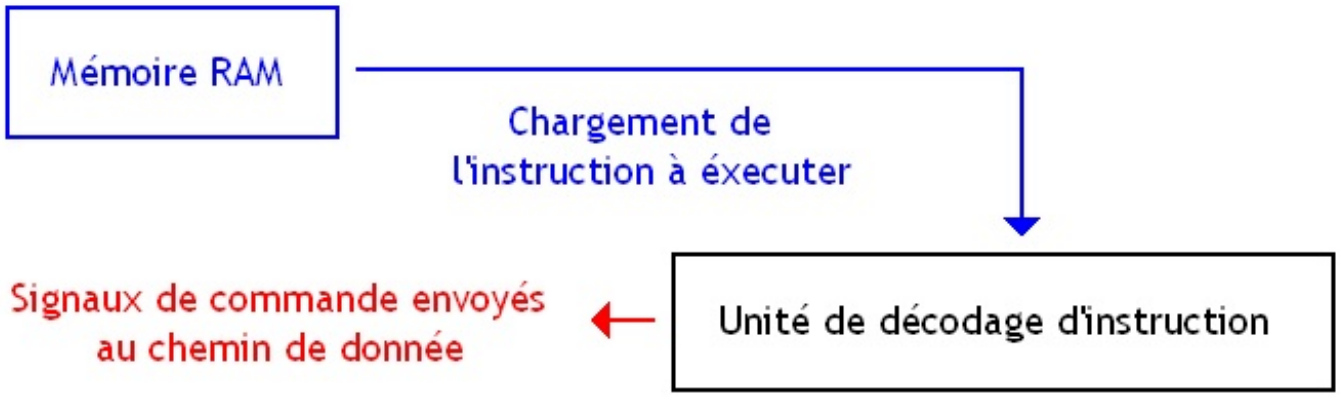
Le séquenceur

Comme on l'a vu plus haut, notre chemin de donnée est remplit d'interrupteurs et de multiplexeurs à configurer. Suivant l'interruption à exécuter, il faut configurer ceux-ci d'une certaine manière pour que notre chemin de donnée soit configuré correctement. Il faut de plus configurer l'entrée de sélection de l'instruction de notre ALU, ainsi que placer ce qu'il faut sur les entrées d'adresses du *register file*.

Il faut donc déduire les bons bits à envoyer sur les entrées correspondantes, en fonction de l'opcode et du mode d'adressage de l'instruction à exécuter. C'est le rôle du **séquenceur** ! Notre séquenceur doit donc gérer le chemin de données pour que celui-ci exécute correctement l'instruction voulue et pas une autre, en tenant compte des différents modes d'adressage et de l'opcode de l'instruction : on dit qu'il décode l'instruction. Cette instruction sera traduite en une suite de micro-opérations, exécutées les unes après les autres. Chaque micro-opération va configurer le chemin de données pour lui faire faire ce qu'il faut.

Pour effectuer une micro-opération, notre séquenceur va donc envoyer des "ordres" qui vont configurer les circuits du chemin de donnée (ALU, *register file*, interrupteurs, multiplexeurs, etc) et leur faire faire ce qu'il faut pour exécuter notre instruction. Ces ordres sont des bits individuels ou des groupes de bits qu'il faut placer sur les entrées des unités de calcul, du *register file*, ou des circuits de gestion du bus interne au processeur (les fameux interrupteurs vus plus haut) dans un ordre bien précis : on les appelle des **signaux de commande**. Lorsque nos unités de calculs (ou d'autres circuits du processeur) reçoivent un de ces signaux de commande, elles sont conçues pour effectuer une action précise et déterminée.

Décoder une instruction est simple : à partir l'opcode et de la partie variable de notre instruction, il faut déduire le ou les signaux de commande à envoyer au chemin de données ou à la *Memory Management Unit*, et parfois déduire l'ordre dans lequel envoyer ces signaux de commande. Pour cela, notre processeur intègre un circuit spécialement dédié à cette étape de décodage : **l'unité de décodage d'instruction**.



Un des signaux de commande généré par notre séquenceur est donc le mot binaire à placer sur l'entrée de sélection d'instruction de l'unité de calcul choisie. Bien sûr, il y a d'autres signaux de commandes à envoyer dans le processeur, mais tout dépend de son architecture et il est très compliqué de faire des généralités sur ce sujet.

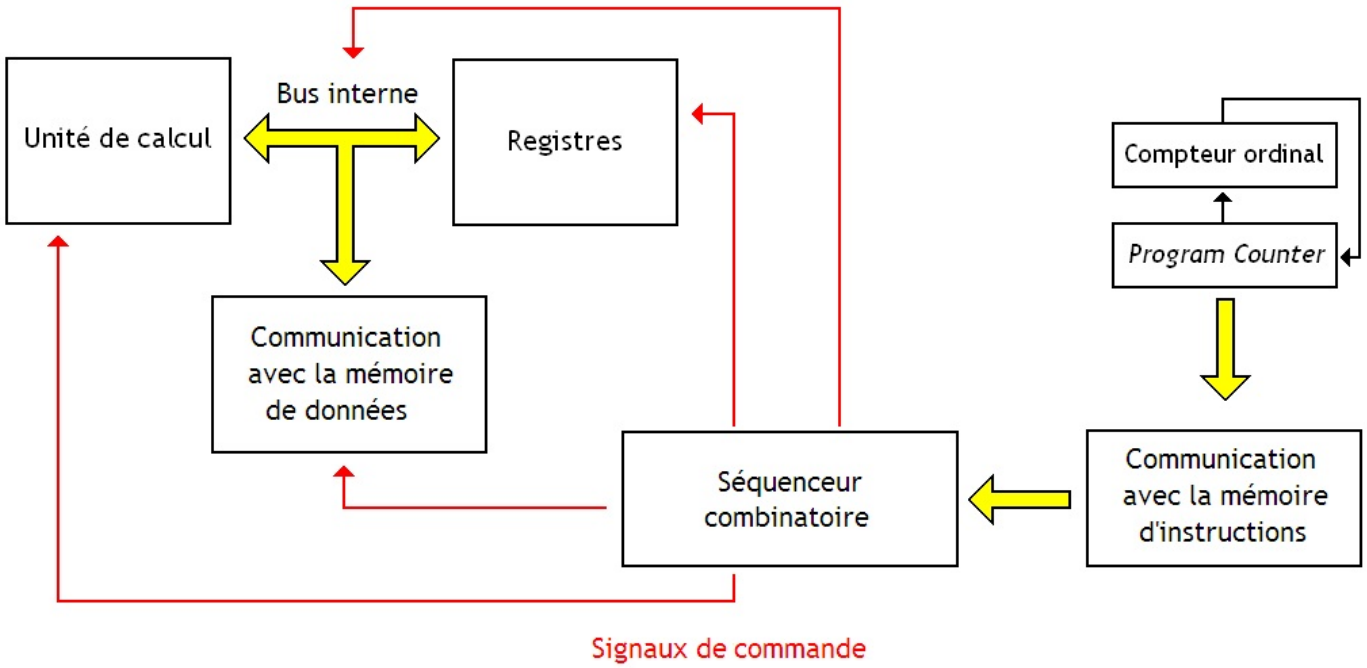
Une des entrées de notre séquenceur est reliée au registre d'instruction, afin d'avoir accès à l'instruction à décoder. Le registre d'état est aussi relié sur une entrée du séquenceur : sans cela, pas de branchements !

Séquenceurs câblés

Il existe des processeurs dans lesquels chaque instruction est décodée par un circuits électronique fabriqué uniquement avec des portes ***ET***, ***NON*** et ***OU*** reliées entre elles : on appelle ce genre de séquenceur un **séquenceur câblé**. Ce genre de séquenceur va se contenter de générer automatiquement les signaux de commande dans le bon ordre pour configurer le chemin de donnée et exécuter notre instruction.

Séquenceur combinatoire

Sur certains processeurs, une instruction s'exécute en une seule micro-opération, chargement depuis la mémoire inclus. C'est très rare, et cela nécessite des conditions assez particulières. Tout d'abord, chaque instruction du processeur ne doit effectuer qu'une seule modification du *Datapath*. Sans cela, on doit effectuer un micro-opération par modification du *Datapath*. Ensuite, la mémoire dans laquelle sont stockées les instructions doit être physiquement séparée de la mémoire dans laquelle on stocke les données. Si ces conditions sont réunies, le séquenceur se résume alors à un simple circuit combinatoire.

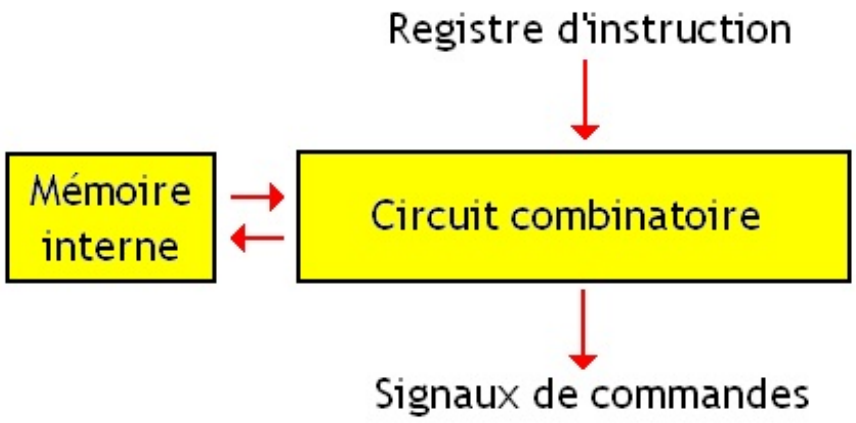


Dans une telle situation, le processeur effectue chaque instruction en un seul cycle d'horloge. Mais autant le dire tout de suite : ces processeurs ne sont pas vraiment pratiques. Avec eux, un accès mémoire prendra autant de temps qu'une addition, ou qu'une multiplication, etc. Pour cela, la durée d'un cycle d'horloge doit se caler sur l'instruction la plus lente. Disposer d'instructions prenant des temps variables permet d'éviter cela : au lieu que toutes nos instructions soient lente, il vaut mieux avoir certaines instructions rapides, et d'autres lentes. Ce qui nécessite d'avoir des instructions découpées en plusieurs micro-opérations.

Séquenceur séquentiel

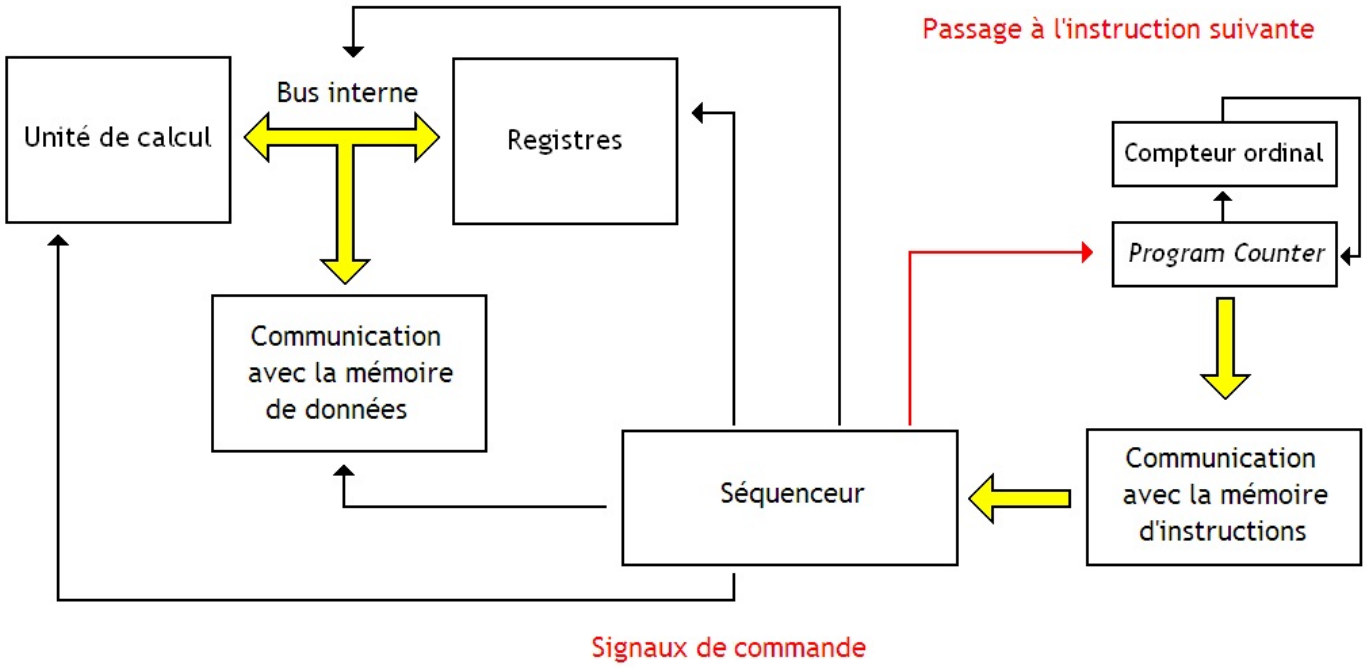
Sur la majorité des processeurs, nos instructions sont découpées en plusieurs micro-opérations, qu'il faudra enchainer dans un certain ordre. Le nombre de micro-opérations peut parfaitement varier suivant l'instruction sans que cela ne pose le moindre problème : certaines seront lentes, d'autres rapides. Certaines instructions prendront plusieurs cycles d'horloge, d'autres non.

Pour enchainer les micro-opérations correctement, notre séquenceur doit savoir à quelle micro-opération il en est rendu dans l'instruction. Il est obligé de mémoriser cette information dans une petite mémoire interne, intégrée dans ses circuits. En conséquence, ces séquenceurs câblés sont obligatoirement des circuits séquentiels. Ils sont composés d'une petite mémoire, et d'un gros circuit combinatoire. Ce circuit combinatoire est chargé de déduire les signaux de commandes en fonction de l'instruction placée en entrée, et du contenu de la mémoire intégrée au séquenceur.

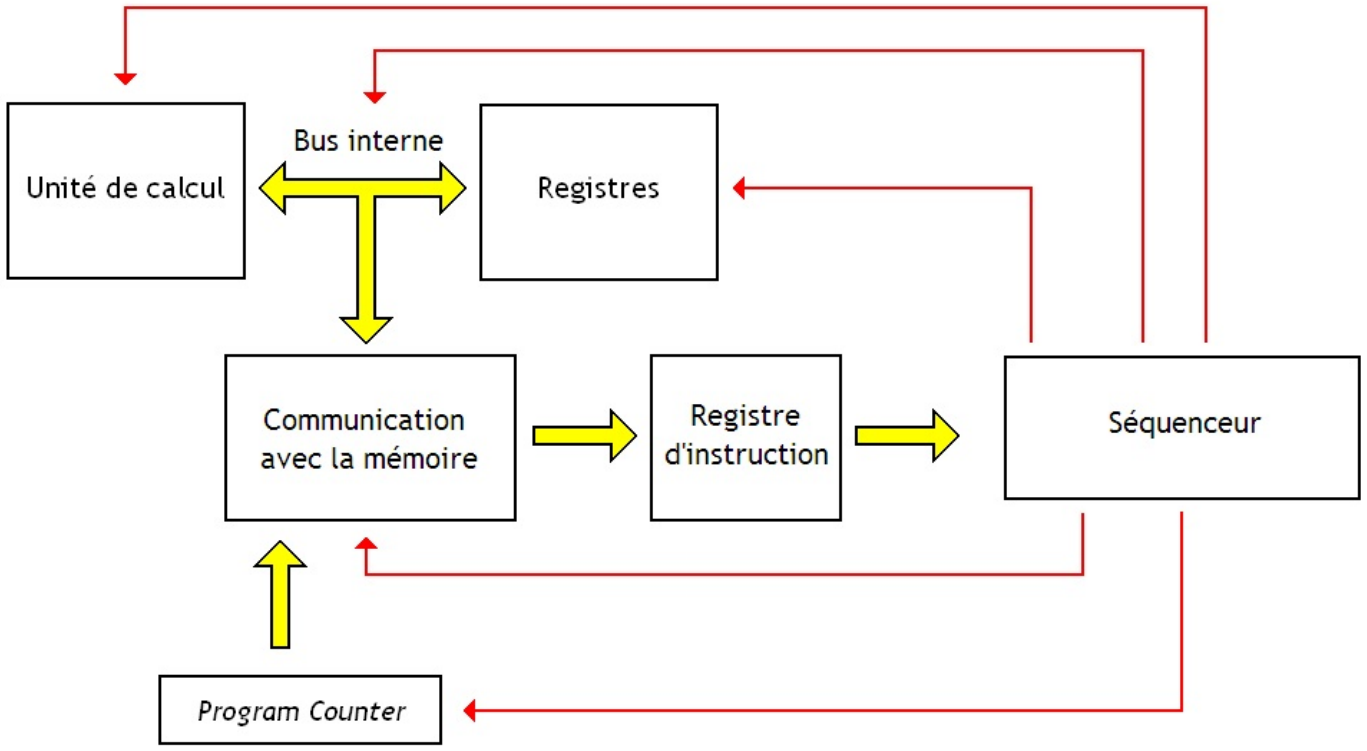


Plus le nombre d'instructions à câbler est important, plus le nombre de portes utilisées pour fabriquer notre séquenceur augmente. Si le nombre d'instructions à câbler est trop grand, on a besoin de tellement de portes que le câblage devient un véritable enfer, sans compter le prix de toutes ces portes qui devient important. Autant dire que les processeurs CISC n'utilisent pas trop ce genre de séquenceurs et préfèrent utiliser des séquenceurs différents. Par contre, les séquenceurs câblés sont souvent utilisés sur les processeurs RISC, qui ont peu d'instructions, pour lequel la complexité du séquenceur et le nombre de portes est assez faible et est supportable.

Mais ce n'est pas la seule conséquence : notre séquenceur doit éviter de changer d'instruction à chaque cycle. Et pour cela, il y a deux solutions, suivant que le processeur dispose de mémoires séparées pour les programme et les instructions ou non. Si la mémoire des instructions est séparée de la mémoire des données, le processeur doit autoriser ou interdire les modifications du *Program Counter* tant qu'il est en train de traiter une instruction. Cela peut se faire avec un signal de commande relié au *Program Counter*.



Mais sur les processeurs ne disposant que d'une seule mémoire (ou d'un seul bus pour deux mémoires), on est obligé de faire autrement. On doit stocker notre instruction dans un registre. Sans cela, pas d'accès mémoire aux données : le bus mémoire serait occupé en permanence par l'instruction en cours d'exécution, et ne pourrait pas servir à charger ou écrire de données.



Séquenceur micro-codé

Créer des séquenceurs câblés est quelque chose de vraiment complexe. On est obligé de déterminer tous les états que peut prendre le circuit, et déterminer toutes les transitions possibles entre ces états. Bref, c'est quelque chose de vraiment compliqué, surtout quand le processeur doit gérer un grand nombre d'instructions machines différentes. Pour limiter la complexité du séquenceur, on a décidé de remplacer le circuit combinatoire intégré dans le séquenceur par quelque chose de moins complexe à fabriquer.

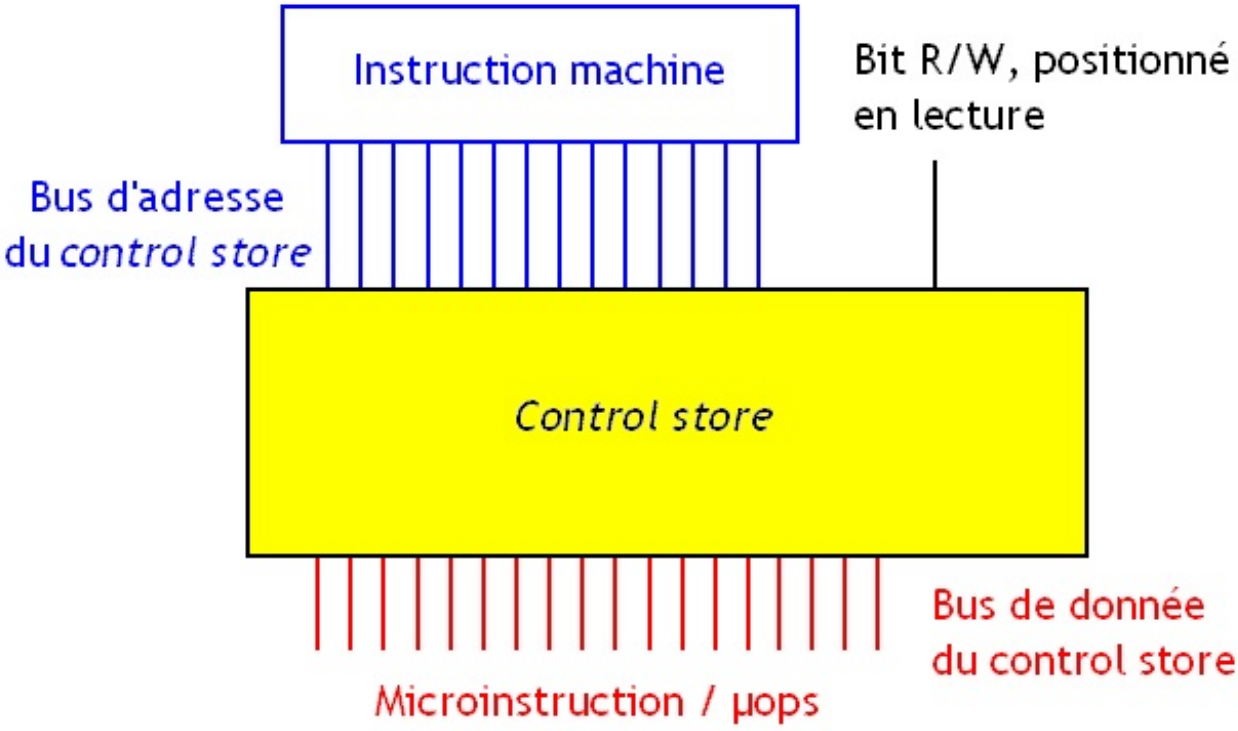
L'idée derrière ce remplacement, c'est que tout circuit combinatoire peut être remplacé par une petite mémoire ROM. Un circuit combinatoire est très simple : il renvoie toujours le même résultat pour des entrées identiques. Dans ces conditions, pourquoi ne pas pré-calculer tout ces résultats et les conserver dans une mémoire ROM ? Cela a un avantage : remplir une mémoire ROM est beaucoup plus simple à faire que créer un circuit combinatoire. Au lieu de déterminer à l'exécution quelle est la suite de micro-opérations équivalente à une instruction machine, on va pré-calculer cette suite de micro-opérations dans notre mémoire ROM. C'est ainsi qu'on a inventé les **séquenceurs micro-codés**.

Control store

Un séquenceur micro-codé contient donc une petite mémoire, souvent de type ROM. Cette mémoire s'appelle le **Control Store**. Cette mémoire va stocker, pour chaque instruction micro-codée, la suite de micro-opérations équivalente. Les suites de micro-opérations contenues dans ce *Control Store* s'appelle le **Micro-code**. le contenu du *control store* est parfois stocké dans une EEPROM et est ainsi modifiable : on peut ainsi changer son contenu et donc modifier ou corriger le jeu d'instruction du processeur si besoin. Idéal pour corriger des bugs ou ajouter des instructions, voire optimiser le jeu d'instruction du processeur si besoin est. On parle alors de *Writeable Control Store*.

Comment notre séquenceur va faire la correspondance entre une instruction micro-codée et la suite de micro-opérations correspondante dans ce *control store* ?

Pour retrouver la suite de micro-opérations correspondante, notre séquenceur considère l'opcode de l'instruction micro-codée comme une adresse. Le *control store* est conçu pour que cette adresse pointe directement sur la suite de micro-opérations correspondante dans la mémoire ROM.



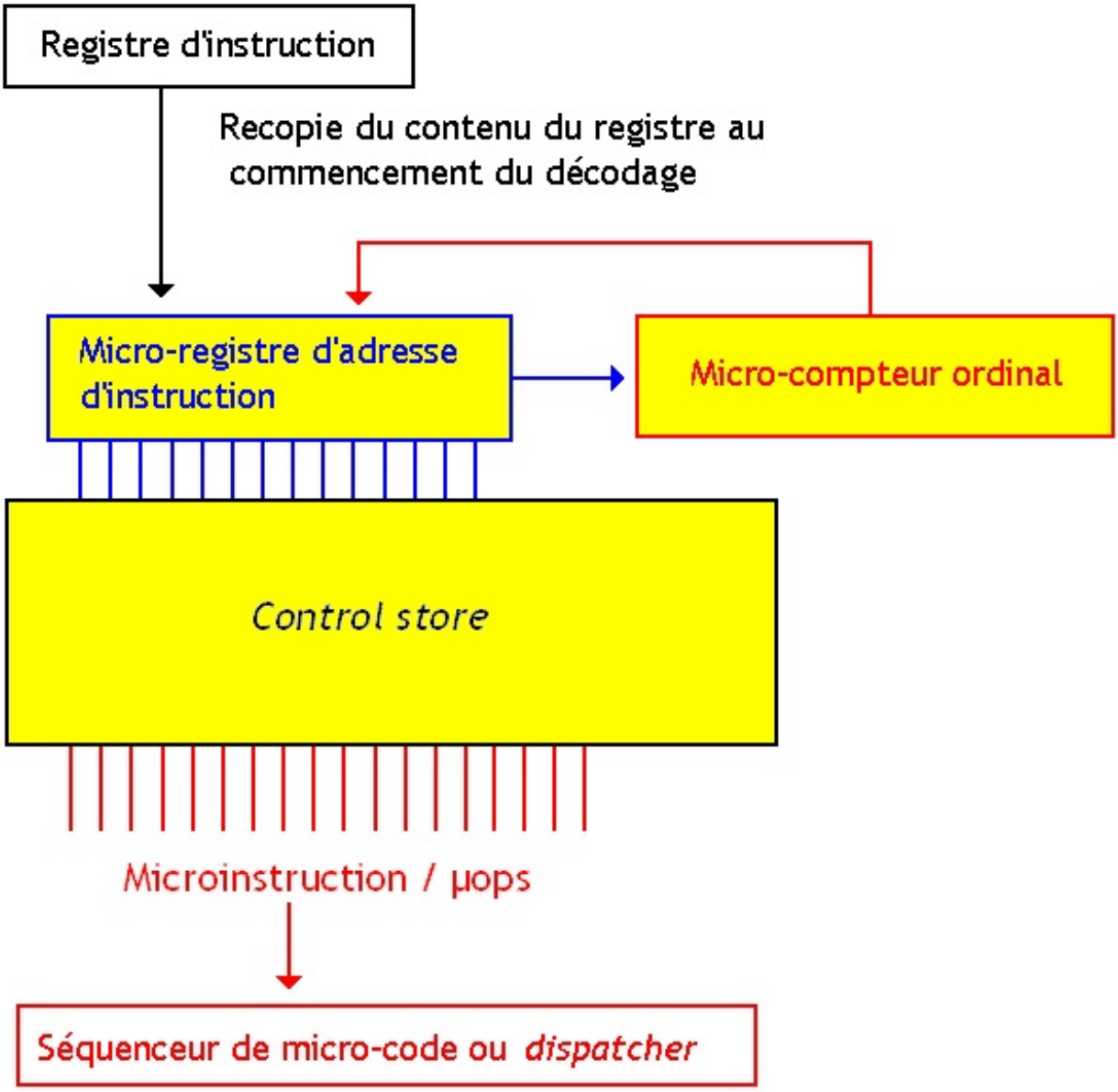
La micro-opération est alors recopiée dans un registre, le **registre de micro-opération**, qui est aux micro-opérations ce que le registre d'instruction est aux instructions machines. Il sert à stocker une micro-opérations pour que le séquenceur puisse décoder celle-ci.

Séquencement du micro-code

Bien sûr, pour ne pas se contenter d'exécuter une seule micro-opération et passer à la suivante, le séquenceur micro-codé contient un **registre d'adresse de micro-opération** qui stocke l'adresse de la micro-opération en cours d'exécution.

Lorsque le décodage d'une instruction machine commence, l'instruction machine, localisée dans le registre d'instruction, est recopiée dans ce micro-registre d'adresse d'instruction. Ce registre d'adresse de micro-opération joue le même rôle que la mémoire qui intégrée dans un séquenceur câblé. Puis, le contenu du registre d'adresse d'instruction est alors augmenté de façon à pointer sur la micro-opération suivante, et ainsi de suite jusqu'à ce qu'on aie exécuté toute la suite de micro-opérations.

Pour cela, ce registre d'adresse de micro-opération est couplé à un **micro-compteur ordinal**, qui augmente l'adresse de ce registre d'adresse de micro-opération de façon à passer à la micro-opération suivante, jusqu'à pointer sur la fin de la suite de micro-opérations correspondant à l'instruction machine. On peut même fabriquer ce micro-compteur ordinal de façon à permettre les branchements entre micro-opérations : une instruction machine peut ainsi être émulée par une boucle de micro-opérations, par exemple.



l'ensemble est appelé le **micro-séquenceur**.

Micro-code horizontal et vertical

Il existe plusieurs sous-types de séquenceurs micro-codés, qui se distingue par la façon dont sont stockées les micro-opérations dans le *Control Store*. On peut mentionner les deux types principaux : ceux qui utilisent un micro-code vertical, et ceux qui utilisent un micro-code horizontal.

Le **micro-code horizontal** est simple : chaque instruction du micro-code (chaque micro-opération) encode directement les

signaux de commande à envoyer aux unités de calcul. Il suffit d'envoyer les bits sur les bons fils pour faire en sorte que le chemin de données fasse ce qu'il faut. Le micro-code horizontal est le plus utilisé de nos jours, du fait de sa simplicité et ses avantages sur le micro-code vertical.

Avec un **micro-code vertical**, ce n'est pas le cas : il faut traduire les micro-opérations en signaux de commande à l'aide d'un séquenceur câblé. Un séquenceur micro-codé utilisant un micro-code vertical est divisé en deux parties : un micro-séquenceur, et une unité de décodage câblée de micro-opérations qui décode les micro-opérations en signaux de commandes.

Le micro-code vertical a un gros désavantage : il faut placer une unité de décodage câblée supplémentaire dans le processeur. Cette unité est malgré tout très simple et utilise peu de portes logiques, ce qui est souvent supportable. Le principal désavantage est le temps mis par cette unité pour traduire une micro-opération en signaux de commande n'est pas négligeable. L'avantage, c'est qu'on peut réduire le nombre de bits utilisés pour chaque micro-opération : il n'est pas rare que les instructions d'un micro-code horizontal fassent plus d'une centaine de bits ! De nos jours, nos processeurs utilisent tous un micro-code horizontal, pour économiser en circuits.

Avantages et inconvénients

Les séquenceurs micro-codés sont plus simples à concevoir : ils n'utilisent pas un grand nombre de portes logiques qu'il faudrait relier entre elles, et cela simplifie beaucoup le travail des concepteurs de processeurs.

De plus, le micro-code est stocké dans une mémoire qu'on peut reprogrammer. Cela a un avantage énorme : on peut conserver le même jeu d'instruction et améliorer les circuits du processeur sans avoir à recréer tout un processeur. Il suffit souvent de récrire le micro-code, sans avoir à refaire tout un séquenceur.

On peut aussi corriger plus facilement les erreurs de conception d'un processeur. Quand on crée un processeur, on commet souvent des erreurs plus ou moins graves. Et bien sachez que ce certains bugs peuvent être corrigés plus facilement avec un séquenceur micro-codé : il suffit de mettre à jour le micro-code. Avec un séquenceur câblé, il faudrait refaire une grande partie du séquenceur ou des unités de calcul, ce qui prendrait un temps fou.

Mais il y a aussi des inconvénients qui ne sont pas négligeables. Un séquenceur micro-codé est plus lent qu'un séquenceur câblé : une mémoire ROM est bien plus lente qu'un circuit combinatoire fabriqué directement avec des portes logiques. Cela se ressent sur la fréquence d'horloge du processeur.

Séquenceurs hybrides

Comme je l'ai dit plus haut, un séquenceur micro-codé est plus économe en transistors et en portes logiques, tandis qu'un séquenceur câblé est plus rapide. Le séquenceurs hybrides sont une sorte de compromis entre ces deux extrêmes : ils sont en partie câblés et en partie micro-codés.

Généralement, une partie des instructions est décodée par la partie câblée du séquenceur, pour plus de rapidité tandis que les autres instructions sont décodées par la partie micro-codée du séquenceur. Cela permet d'éviter de câbler une partie du séquenceur qui prendrait beaucoup de portes pour décoder des instructions complexes, généralement rarement utilisées, tout en gardant un décodage rapide pour les instructions simples, souvent utilisées.

Parfois, cette technique est adaptée en scindant les parties câblées et les parties micro-codées en plusieurs unités de décodage d'instructions bien séparées. Notre processeur contient ainsi plusieurs unités de décodage d'instruction, l'une d'entre elle étant câblée, et l'autre est micro-codée.

Les *transport triggered architectures*

Sur certains processeurs, les instructions machines sont très simples et correspondent directement à des micro-instructions qui agissent sur le bus. En clair, toutes les instructions machines permettent de configurer directement le bus interne au processeur, et il n'y a pas de séquenceur ! De tels processeurs sont ce qu'on appelle des ***transports triggered architectures***.

Sur ces processeurs, on ne peut donc que configurer le bus. Pire : le bus est organisé de façon à ce qu'on ne puisse pas avoir accès aux unités de calcul directement : on doit passer par des registres intermédiaires spécialisés dont le but est de stocker les opérandes d'une instruction. Tout ce qu'on peut faire, c'est connecter des registres sur le bus pour leur faire échanger des données et communiquer avec la mémoire. On pourrait se dire que rien ne permet d'effectuer d'instructions d'additions ou d'opérations de ce genre et que ces processeurs sont un peu inutiles. Mais ce n'est pas le cas ! En fait, ces processeurs rusent.

Ces processeurs contiennent des registres spéciaux, reliés à une unité de calcul. Ces registres servent spécialement à stocker les opérandes d'une instruction machine, tandis que d'autres servent aussi à déclencher des instructions : lorsqu'on écrit une donnée dans ceux-ci, cela va automatiquement déclencher l'exécution d'une instruction bien précise par l'unité de calcul, qui lira le contenu des registres chargés de stocker les opérandes.

Par exemple, un processeur de ce type peut contenir trois registres ***Add.opérande.1***, ***Add.déclenchement*** et ***Add.résultat***. Le premier registre servira à stocker la première opérande de l'addition. Pour déclencher l'opération d'addition, il suffira d'écrire la seconde opérande dans le registre ***Add.trigger***, et l'instruction s'exécutera automatiquement. Une fois l'instruction terminée, le résultat de l'addition sera automatiquement écrit dans le registre ***Add.resultat***. Il existera des registres similaires pour la multiplication, la soustraction, les comparaisons, etc.

Sur certains de ces processeurs, on a besoin que d'une seule instruction qui permet de copier une donnée d'un emplacement (registre ou adresse mémoire) à un autre. Pas d'instructions `Load`, `Store`, etc : on fusionne tout en une seule instruction supportant un grand nombre de modes d'adressages. Et donc, on peut se passer complètement d'opcode, vu qu'il n'y a qu'une seule instruction : pas besoin de préciser quelle est celle-ci, on le sait déjà. Sympa, non ?

L'utilité de ces architectures n'est pas évidente. Leur raison d'exister est simplement la performance : manipuler le bus interne au processeur directement au lieu de laisser faire les circuits du processeur permet de faire pas mal de raccourcis et permet quelques petites optimisations. On peut ainsi travailler directement avec des micro-instructions au lieu de devoir manipuler des instructions machines, ce qui permet parfois de ne pas utiliser de micro-instructions en trop. Mais l'intérêt est assez faible.

L'étape de fetch

On a donc nos unités de calcul bien fonctionnelles, bien comme il faut. Notre processeur ne peut malgré tout pas encore effectuer d'instructions. Ben oui : on doit gérer l'étape de *Fetch*. Cette étape est assez simple, et est effectuée de la même façon sur tous les processeurs.

Elle est décomposée en trois grandes étapes :

- envoyer le contenu du *Program Counter* sur le bus d'adresse ;
- récupérer l'instruction sur le bus de donnée et la copier dans un registre qu'on appelle le registre d'instruction ;
- modifier le *Program Counter* pour pointer sur l'instruction suivante.

Il faut noter que certains processeurs vont simultanément récupérer l'instruction sur le bus de donnée et modifier le *Program Counter*. Il faut dire que ces deux étapes sont indépendantes : elles ne touchent pas aux même registres et n'utilisent pas les mêmes circuits. On peut donc effectuer ces deux étapes en parallèles, sans aucun problème. Les deux premières étapes sont assez simples à effectuer : il s'agit d'une simple lecture, qui peut être prise en charge par notre *Datapath*. Mais la troisième est un peu plus intrigante, et peut être implémentée de plusieurs façons différentes.

Registre pointeur instruction

L'étape de ***fetch*** consiste à copier l'instruction à exécuter dans le registre d'instruction. Pour savoir où est cette instruction, le processeur stocke son adresse dans un registre. Comme vous le savez, ce registre s'appelle le **registre d'adresse d'instruction**. Pour charger notre instruction, il suffit de recopier le contenu du registre d'adresse d'instruction sur le bus d'adresse (en passant éventuellement par un registre d'interfaçage mémoire), configurer le bus de commande pour effectuer une lecture, et connecter le bus de donnée sur le registre d'instruction (en passant éventuellement par un registre d'interfaçage mémoire). Le séquenceur le fait automatiquement, en envoyant les signaux de commande adéquats.

Au démarrage d'un programme, notre processeur doit exécuter la première instruction, placée à une certaine adresse dans la mémoire programme. Il suffit d'initialiser le registre pointeur d'instruction à cette adresse pour commencer au bon endroit. Naïvement, on peut penser que la première adresse de la mémoire programme est l'adresse **0**, et qu'il suffit d'initialiser le registre pointeur d'instruction à **0**. Mais c'est parfois faux : sur certains processeurs, les premières adresses sont réservées et servent à adresser des trucs assez importants, comme la pile ou le vecteur d'interruptions. L'adresse de départ de notre programme n'est pas l'adresse **0**, mais une autre adresse. Dans ces cas-là, le processeur est conçu pour initialiser le registre pointeur d'instruction à la bonne adresse.

Juste une remarque : les processeurs haute performance modernes peuvent charger plusieurs instructions qui se suivent en une seule fois et copier le tout dans le registre d'instruction. Cela permet d'éviter d'accéder trop souvent à la mémoire.

Compteur ordinal

On sait donc où est localisée notre instruction dans la mémoire. C'est super, une partie du problème est résolue. Reste la seconde partie.

Et l'instruction suivante, c'est quoi son adresse ?

Aie ! Les ennuis commencent. Exécuter une suite d'instructions sans savoir où est la suivante risque d'être un peu compliqué.

Pour répondre : on ne sait pas où est la prochaine instruction, mais on peut le calculer !

Le fait que les instructions soient toutes stockées dans l'ordre dans la mémoire nous arrange bien. Pour calculer l'adresse suivante facilement, il faut que toutes les instructions soient placées les unes à coté des autres en mémoire et surtout qu'elles y soient classées dans l'ordre dans lesquelles on veut les exécuter. Quand on dit "classées les unes à coté des autres et dans l'ordre", ça veut dire ceci :

Adresse	Instruction
1	Charger le contenu de l'adresse 0F05
2	Charger le contenu de l'adresse 0555
3	Additionner ces deux nombres
4	Charger le contenu de l'adresse 0555
5	Faire en XOR avec le résultat antérieur
...	...
5464	Instruction d'arrêt

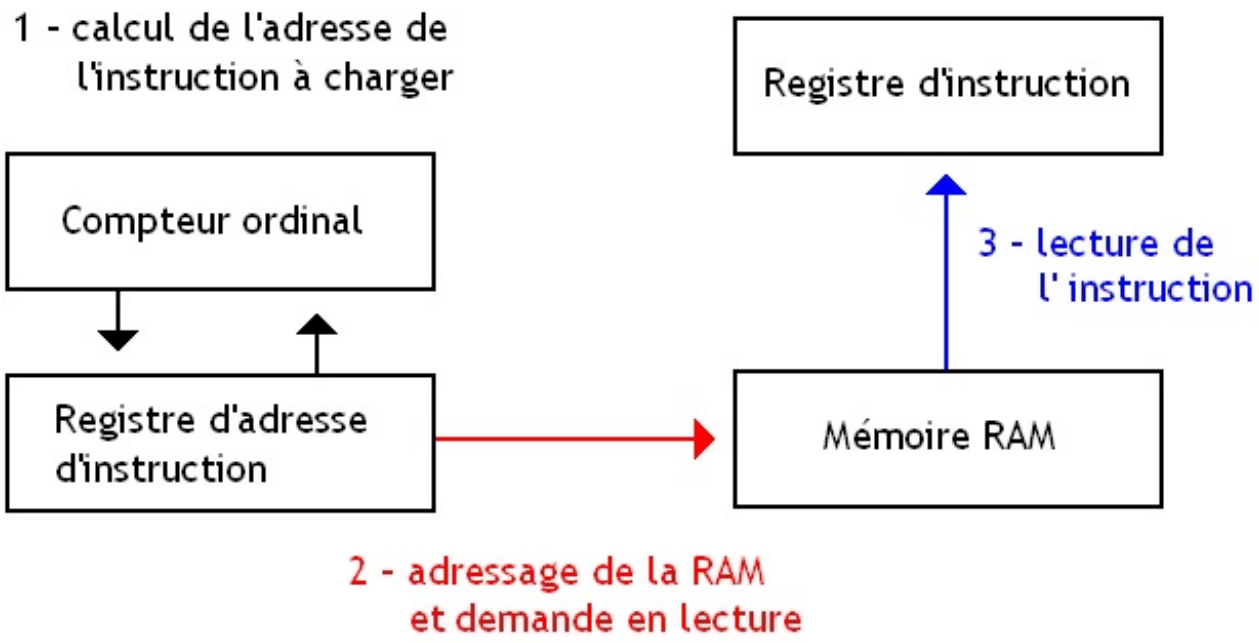
Dans cet exemple, une instruction peut être stockée dans une seule adresse. En fait, ce n'est pas toujours le cas : parfois, une instruction peut tenir dans plusieurs cellules mémoire, mais le principe reste le même.

Avec ce genre d'organisation des instructions en mémoire, on peut alors calculer l'adresse de l'instruction suivante en mémoire avec des additions. Cette addition peut être effectuée de deux manières : soit on utilise notre ALU pour effectuer cette addition, soit on délègue cette tache à un circuit spécialisé.

Avec la première solution, il suffit de rajouter une étape pour exécuter notre instruction : en plus des étapes de *Fetch*, de décodage, et les autres, dépendantes du mode d'adressage ou de l'instruction, on rajoute une ou plusieurs étapes qui vont modifier le registre d'adresse d'instruction pour le faire pointer sur l'instruction suivante. Le registre d'adresse d'instruction est relié au chemin de donnée et le calcul de l'adresse suivante est ainsi réalisée par l'ALU.

Avec la seconde solution, le calcul de l'adresse de l'instruction suivante est le rôle d'un petit circuit nommé le **compteur ordinal**. Ainsi, pas besoin de rajouter des étapes supplémentaires pour effectuer notre calcul d'adresse : celui-ci est effectué en parallèle de l'exécution de notre instruction, automatiquement, sans avoir à utiliser le chemin de donnée. Cette solution demande de rajouter un circuit, mais ce circuit est assez rudimentaire, et ne prend presque pas de place : on a tout à gagner à utiliser cette deuxième solution sur de gros processeurs.

Une fois cette adresse calculée, il suffira de l'envoyer sur le bus d'adresse, configurer le bus de commande en lecture, et connecter le registre d'instruction sur le bus de donnée pour la charger dans le registre d'instruction.



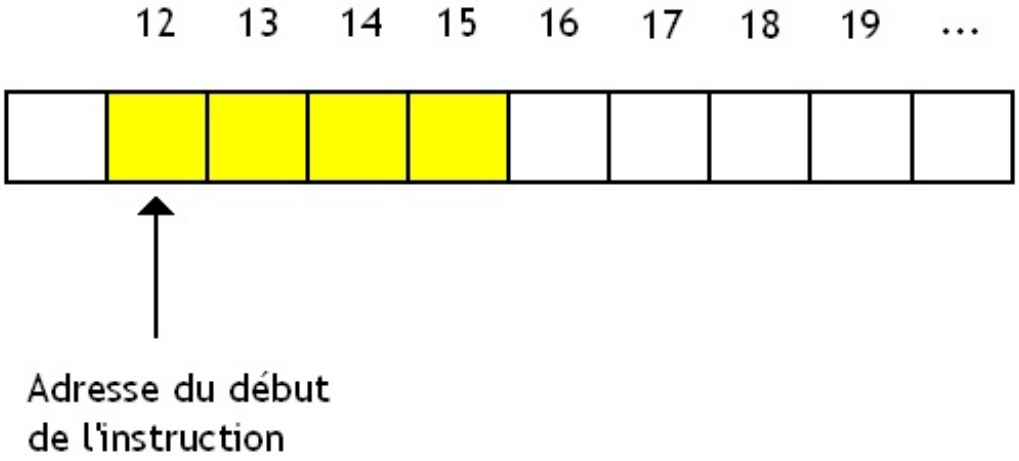
Le calcul de l'adresse suivante

On sait maintenant comment est organisée notre unité de *fetch*. Il serait maintenant intéressant de chercher à savoir comment notre compteur ordinal va faire pour calculer l'adresse de l'instruction suivante. Sachez que la méthode varie suivant le processeur et suivant la longueur de l'instruction.

Mais dans tous les cas, elle se base sur un principe simple : l'instruction suivante est immédiatement après l'instruction en cours. On peut donc calculer cette prochaine adresse via un calcul super simpliste.

Adresse de la prochaine instruction = Adresse de l'instruction en cours + Longueur de l'instruction en cours

Exemple : l'instruction en cours est stocké à l'adresse 12, et fait 4 bytes de long. On voit bien que l'instruction suivante est placée à l'adresse 12 + 4 (ce qui fait 16). L'adresse de l'instruction suivante est donc égale à l'adresse de l'instruction en cours, plus la longueur de cette instruction.



L'adresse de l'instruction en cours est connue : elle est stockée dans le registre d'adresse d'instruction avant sa mise à jour. Reste à connaitre la longueur cette instruction.

Instructions de tailles fixes

Calculer l'adresse de l'instruction suivante est très simple lorsque les instructions ont toutes la même taille. La longueur étant connue, on sait d'avance quoi ajouter à l'adresse d'une instruction pour obtenir l'adresse de l'instruction suivante.

Notre compteur ordinal peut ainsi être conçu assez simplement, en utilisant un vulgaire circuit combinatoire encore plus simple qu'un additionneur.

Instructions de tailles variables

Mais cette technique ne marche pas à tous les coups : certains processeurs CISC ont des instructions qui ne sont pas de taille fixe, ce qui complique le calcul de l'adresse de l'instruction suivante.

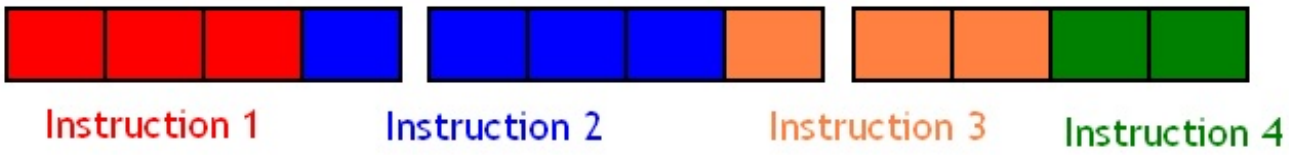
Il y a plusieurs solutions à ce problème. La plus simple consiste à indiquer la longueur de l'instruction dans une partie de l'opcode ou de la représentation en binaire de l'instruction. Une fois cette longueur connue, on effectue alors l'addition avec le contenu du registre d'adresse d'instruction et puis c'est fini.

Une autre solution consiste simplement à charger notre instruction morceaux par morceaux et rassembler le tout une fois que tous les morceaux ont étés chargés. Les morceaux sont copiés dans le registre d'instruction les uns à la suite des autres. Quand la totalité de l'instruction est disponible, le processeur envoie l'instruction vers les circuits chargés de décoder l'instruction. Vu que chaque morceau a une taille fixe, le compteur ordinal est incrémenté à chaque cycle d'horloge de la taille d'un morceau, exprimée en nombre de cases mémoires. Le seul défaut de cette approche, c'est qu'il faut trouver un moyen de savoir si une instruction complète à été chargée ou pas : un nouveau circuit est requis pour cela.

Et enfin, il existe une dernière solution, qui est celle qui est utilisée dans vos processeurs, ceux qu'on trouve dans les processeurs haute performance de nos PC. Avec cette méthode, on charge un gros bloc de bytes qu'on découpe progressivement en instructions, en déduisant leurs longueurs au fur et à mesure. Généralement, ce découpage se fait instruction par instruction : on sépare la première instruction du reste du bloc et on l'envoie à l'unité de décodage, puis on recommence jusqu'à atteindre la fin du bloc.

Généralement, la taille de ce bloc est conçue pour être de la même longueur que l'instruction la plus longue du processeur. Ainsi, on est sur de charger obligatoirement au moins une instruction complète, et peut-être même plusieurs, ce qui est un gros avantage.

Cette solution pose quelques problèmes : il se peut qu'on n'aie pas une instruction complète lorsque l'on arrive à la fin du bloc, mais seulement un morceau. Imaginez par exemple le cas où un bloc 4 octets. On peut se retrouver dans des situations comme celle-ci :



La dernière instruction déborde : elle est à cheval entre deux blocs. Dans ce cas, on a pas trop le choix : on doit charger le prochain bloc avant de faire quoique ce soit. Ceci dit, au delà de ce petit inconvénient, cette technique a un gros avantage : on peut parfaitement charger plusieurs instructions en une fois, si elles sont regroupées dans un seul bloc. Et cela arrive très souvent : on évite de nombreux accès à la mémoire.

Les instructions qui posent problème

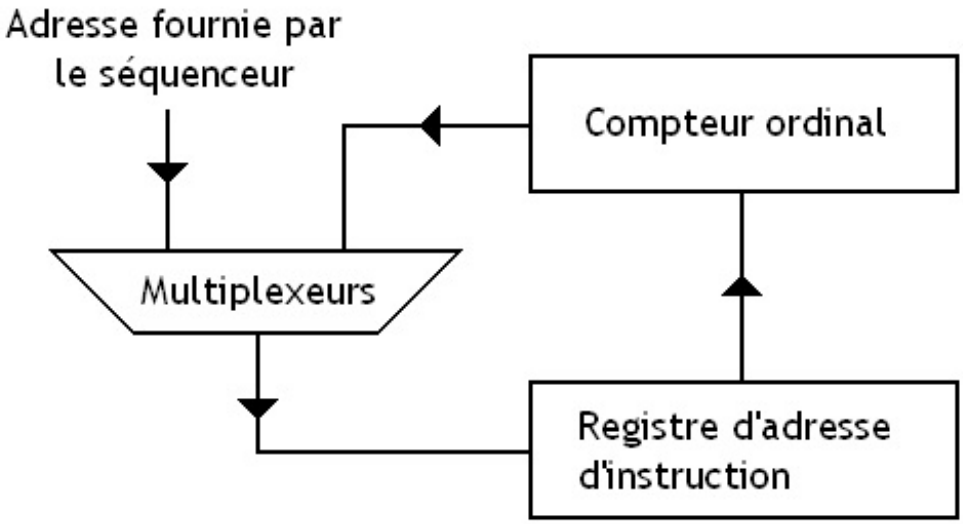
Dans de rares cas, pour certaines instructions et sur quelques rares architectures, on peut stopper le fonctionnement du compteur ordinal tant qu'une condition n'est pas remplie. Le compteur ordinal étant "stoppé", les contenus du registre d'adresse d'instruction et du registre d'instruction ne sont pas modifiés : on exécute donc la même instruction en boucle. On peut citer en exemple les instructions de traitement des chaines de caractères de l'instruction x86, qui peuvent être répétées ainsi.

Les branchements

Lors d'un branchement, l'adresse de destination du branchement va être copiée dans le registre d'adresse d'instruction. Pour cela, il va falloir permettre l'accès en écriture dans ce registre, sous certaines conditions.

Pour les branchements directs (dont l'adresse est fournie par l'instruction), on permet au séquenceur de fournir l'adresse de destination sur une de ses sorties reliée plus ou moins indirectement sur notre registre d'adresse d'instruction. Pour les branchements indirects (ceux dont l'adresse de destination est stockée dans un registre), il suffit de relier le registre d'adresse d'instruction et le registre contenant l'adresse de destination du branchement via le bus interne au processeur. Il faudra alors

choisir entre l'adresse calculée par notre compteur ordinal et l'adresse fournie par le séquenceur, suivant l'occurrence d'un branchement ou non. Ce choix est réalisé par un certain nombre de multiplexeurs.



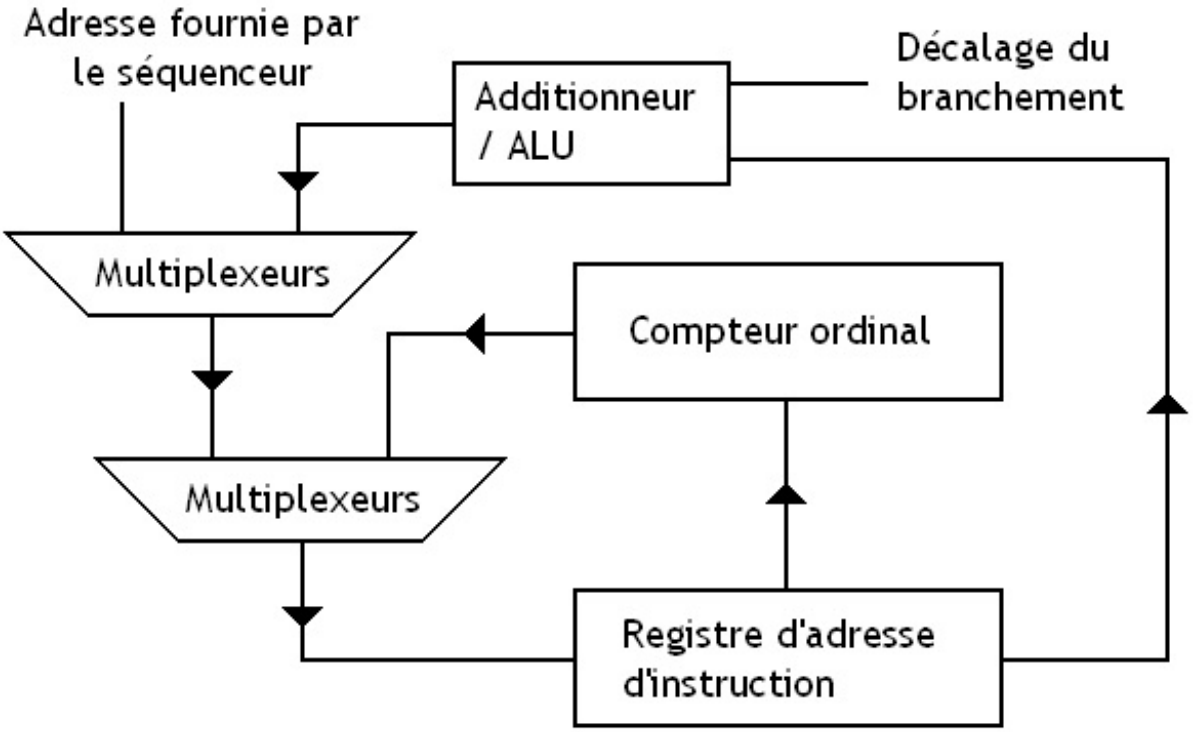
Comme vous le voyez sur ce schéma, s'il n'y a pas de branchement, le multiplexeur se contente de relier la sortie du compteur ordinal sur le registre d'adresse d'instruction. On va donc écrire cette adresse calculée dans le registre d'adresse d'instruction et on va directement passer à l'instruction suivante.

En cas de branchement, le multiplexeur va relier la sortie du séquenceur ou le registre (en cas de branchements indirects) dans lequel se trouve l'adresse du branchement : l'adresse calculée par notre compteur ordinal est perdue et on se retrouve avec l'adresse de notre branchement dans le registre d'adresse d'instruction. C'est le séquenceur qui configure correctement le multiplexeur comme il faut, suivant l'occurrence d'un branchement ou non.

Les branchements relatifs

Je ne sais pas si vous vous en souvenez, mais il existe un certain type de branchement qu'on a vu précédemment : les branchements relatifs. Pour rappel, ces branchements demandent au processeur de brancher vers une adresse située à une certaine distance par rapport à l'adresse de l'instruction en cours d'exécution. Ces branchements permettent de localiser un branchement par rapport à l'instruction en cours d'exécution : par exemple, cela permet de dire "le branchement est 50 adresses plus loin".

Pour prendre en compte ces branchements, on a encore une fois deux solutions. On peut réutiliser l'ALU pour calculer l'adresse de notre branchement. Mais on peut aussi faire autrement : il suffit juste de rajouter un petit circuit qui va alors additionner ce fameux décalage à l'adresse de l'instruction en cours, qui est bien évidemment stockée dans le registre d'adresse d'instruction, et rajouter un multiplexeur pour se charger de sélectionner l'adresse calculée ainsi quand il le faut.



C'est le séquenceur qui configure correctement les multiplexeurs comme il faut, et qui fourni le décalage, suivant l'occurrence d'un branchement ou non, et suivant la localisation de l'adresse de destination (registre ou fournie par l'instruction).

On remarque aussi qu'on pourrait faire cette addition du décalage en utilisant l'ALU : il suffit de relier notre registre d'adresse d'instruction sur l'entrée et la sortie de l'ALU (mais pas par les même ports), et d'envoyer le décalage sur une autre entrée : tout se passe comme si l'on utilisait le mode d'adressage immédiat pour le décalage, et le mode d'adressage implicite pour le registre d'adresse d'instruction.

Branchements indirects

Enfin, il faut encore gérer les branchements indirects. Pour cela, il suffit simplement de relier notre registre d'adresse d'instruction sur un bus interne du processeur (via le chemin de donnée, donc). On peut alors copier le contenu d'un registre dans notre registre d'adresse d'instruction, ce qui est exactement ce que fait un branchement indirect.

L'exception qui confirme la règle

Attention, je vais vous faire une révélation : certains processeurs ne contiennent pas de compteur ordinal et sont incapables de calculer la prochaine adresse. Ces processeurs utilisent une autre méthode pour indiquer au processeur quelle sera la prochaine instruction à exécuter : **chaque suite de bits représentant une instruction contient l'adresse mémoire de la prochaine instruction à exécuter**.

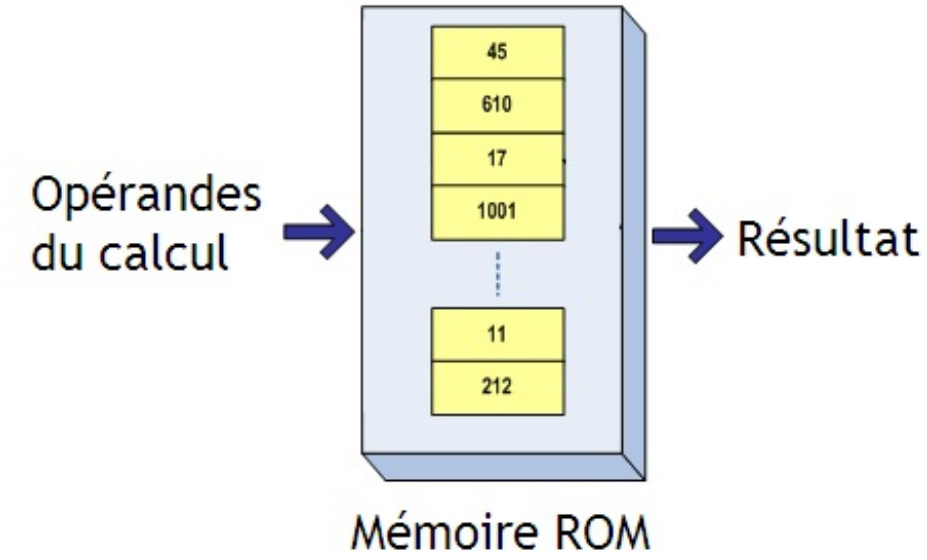
Code opération	Opérande 1	Opérande 2	Adresse de l'instruction suivante

Les processeurs de ce type contiennent toujours un registre d'adresse d'instruction : la partie de l'instruction stockant l'adresse de la prochaine instruction est alors recopiée dans ce registre, pour faciliter sa copie sur le bus d'adresse. Mais le compteur ordinal n'existe pas. Sur des processeurs aussi bizarres, pas besoin de stocker les instructions en mémoire dans l'ordre dans lesquelles elles sont censées être exécutées. Mais ces processeurs sont très très rares et peuvent être considérés comme des exceptions qui confirment la règle.

Les circuits d'une ALU entiére

Après avoir vu l'architecture extérieure d'un processeur, on va pousser le vice dans ses deniers retranchements et descendre au niveau des circuits. Nous allons voir comment fabriquer une ALU. Nous allons donc commencer par voir des circuits permettant d'effectuer des additions, des multiplications, des divisions, des décalages, et des soustractions. Tous ces circuits sont des circuits combinatoires, c'est à dire qu'ils donnent toujours le même résultat pour des entrées identiques. En quelque sorte, ils n'ont pas de mémoire.

Mais pour commencer, j'ai une petite révélation à vous faire : une unité de calcul peut être conçue en utilisant uniquement une mémoire ROM. Oui, vous avez bien lus ! Le fait est que tout circuit combinatoire peut être implémenté en utilisant une mémoire ROM, c'est à dire une mémoire dans laquelle on ne peut pas écrire, et qui ne s'effacent pas quand on coupe le courant. Et notre ALU n'échappe pas à cette règle. Il suffit simplement de pré-calculer les résultats de nos opérations, et de les stocker dans cette mémoire. En envoyant les opérandes de nos instructions sur le bus d'adresse de cette ROM, on sélectionne le bon résultat, et on n'a plus qu'à le récupérer sur la sortie.



Le seul problème, c'est qu'avec cette technique, la taille de la mémoire augmente un peu trop vite pour pouvoir tenir dans un processeur. Par exemple, si je veux pré-calculer tous les résultats d'une addition effectuée sur deux nombres de 32 bits, j'aurais besoin d'une mémoire ROM plus grosse que votre disque dur ! Autant dire que ce genre de technique ne marche que pour des calculs dont les opérandes sont codées sur très peu de bits. Pour le reste, on va devoir créer des circuits capables d'effectuer nos calculs.

On pourrait penser utiliser les méthodes vues au chapitre 3. Mais les tables de vérité qu'on aurait à écrire seraient démesurément grandes. Créer une ALU 32 bits nécessiterait des tables de vérité comprenant plus de 4 milliards de lignes ! A la place, nous allons devoir ruser...

Décalages et rotations

On va commencer par les circuits capables d'exécuter des instructions de décalage et de rotation. On en a très brièvement parlé dans les chapitres précédents, mais vous ne savez peut-être pas ce qu'elles font. Vu la situation, une petite explication sur ces instructions ne fera pas de mal.

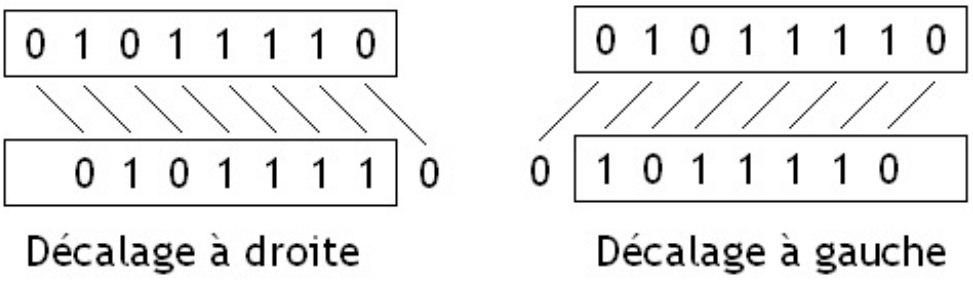
Décalages et rotations

Il existe plusieurs types de décalages, dont deux vont nous intéresser particulièrement : les décalages logiques, et les décalages arithmétiques.

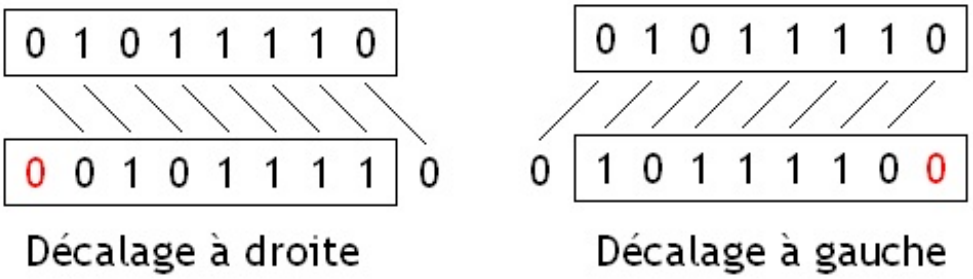
Logical shift

Le décalage logique, aussi appelé *logical shift*, est le décalage le plus simple à comprendre. Effectuer un décalage logique sur un nombre consiste simplement à décaler tout ses chiffres d'un ou plusieurs crans vers la gauche ou la droite.

Exemple avec un décalage d'un cran.



Comme vous le voyez, vu que nos nombres sont de taille fixe, certains bits sortent du nombre. Pour le décalage à droite, c'est le bit de poids faible qui sort du nombre, tandis que pour le décalage à gauche, c'est le bit de poids fort qui est perdu. On remarque aussi que certains bits du résultat sont "vides" : ils ne peuvent pas être déduits de la valeur du nombre à décaler. Ces vides sont remplis par des zéros.

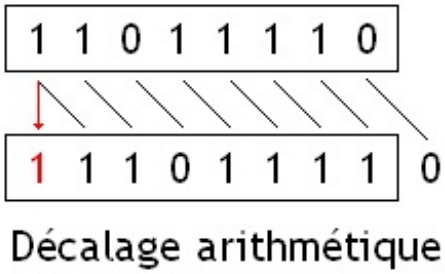


Grâce à ce remplissage par des zéros, un décalage vers la gauche d'un rang est équivalent à une multiplication par 2 pour des entiers non-signés ou pour des entiers signés positifs. Même chose pour le décalage vers la droite qui revient à diviser un nombre entier par 2. Avec des nombres signés, ce n'est pas le cas : on obtient un résultat qui n'a pas grand sens mathématiquement. De même, pour des entiers non-signés ou positifs, on peut généraliser avec un décalage de n rangs vers la droite/gauche : un tel décalage correspond à une multiplication ou division entière par 2^n. Cette propriété est souvent utilisée par certains compilateurs, qui préfèrent utiliser des instructions de décalages (qui sont des instructions très rapides) à la place d'instructions de multiplication ou de division qui ont une vitesse qui va de moyenne (multiplication) à particulièrement lente (division).

Il faut remarquer un petit détail : lorsqu'on effectue une division par 2^n - un décalage à droite -, certains bits de notre nombre vont sortir du résultat et être perdus. Cela a une conséquence : le résultat est tronqué ou arrondi. Plus précisément, le résultat d'un décalage à droite de n rangs sera égal à la partie entière du résultat de la division par 2^n.

Arithmetical shift

Pour pouvoir effectuer des divisons par 2^n sur des nombres négatifs en utilisant un décalage, on inventé les décalages arithmétiques ou *arithmétical shift*. Ces décalages sont similaires aux *logical shift*, à un détail prêt : pour les décalages à droite, le bit de signe de notre nombre n'est pas modifié, et on remplit les vides laissés par le décalage avec le bit de signe.

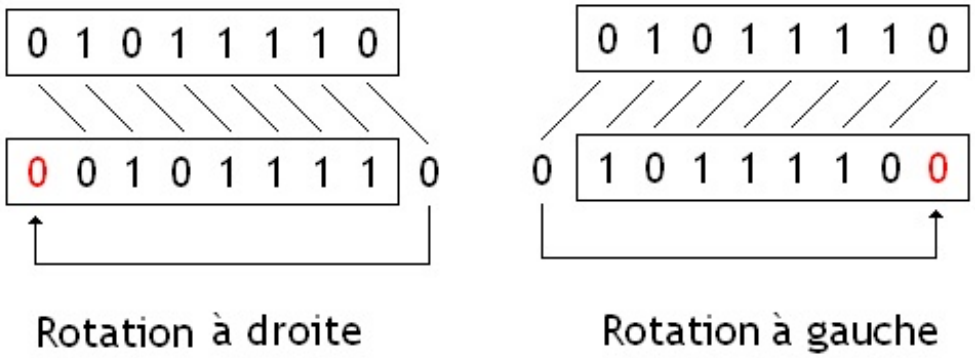


Ces instructions sont équivalentes à une multiplication/division par 2^n, que le nombre soit signé ou non, à un détail prêt : l'arrondi n'est pas fait de la même façon pour les nombres positifs et négatifs. Cela pose un problème avec les nombres codés en complément à deux (ceux codés en complément à un ne sont pas concernés).

Pour les nombres positifs ou nuls, un *arithmétical shift* donne toujours le même résultat qu'un *logical shift* et on n'a pas de problème : le résultat est arrondi vers zéro quelque soit le décalage. Mais la situation change pour les nombres négatifs qui sont arrondis vers moins l'infini. Pour donner un exemple, $\frac{9}{2}$ sera arrondi en 4, tandis que $\frac{-9}{2}$ sera arrondi en -5. Pour éviter tout problème, on peut corriger le résultat en utilisant quelques instructions supplémentaires. Mais cela reste au minimum 6 à 26 fois plus rapide que d'effectuer la division.

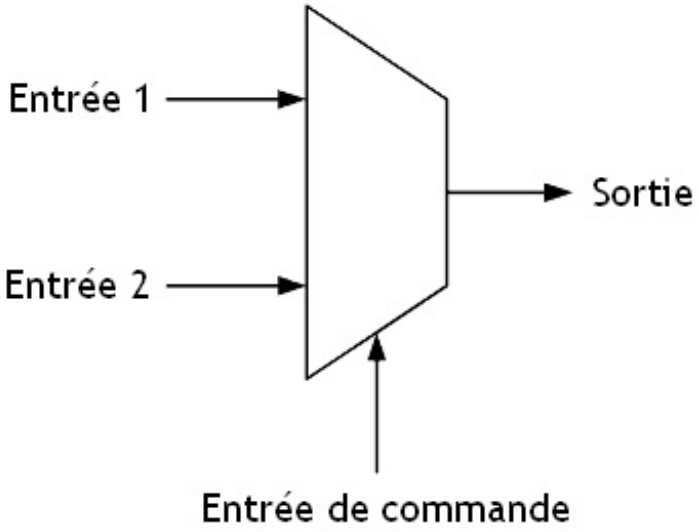
Rotations

Les instructions de rotation sont similaires aux *logical shift*, à part que les bits qui sortent du nombre d'un coté rentrent de l'autre et servent à boucher les trous.



Multiplexeurs

Pour commencer, les circuits capables d'effectuer des décalages et des rotations sont fabriqués avec des composants électroniques qu'on appelle des **multiplexeurs**, aussi appelés des **MUX**. Un multiplexeur possède plusieurs entrées et une sortie. Par plusieurs entrées, on veut dire que suivant le multiplexeur, le nombre d'entrées peut varier. Le rôle d'un multiplexeur est de recopier le contenu d'une des entrées sur sa sortie. Bien sûr, il faut bien choisir l'entrée qu'on veut recopier sur la sortie : pour cela, notre multiplexeur contient une entrée de commande qui permet de spécifier quelle entrée doit être recopiée. Dans la suite de ce chapitre, on ne va utiliser que des multiplexeurs qui possèdent deux entrées et une sortie. Après tout, nous travaillons en binaire, n'est-ce pas.

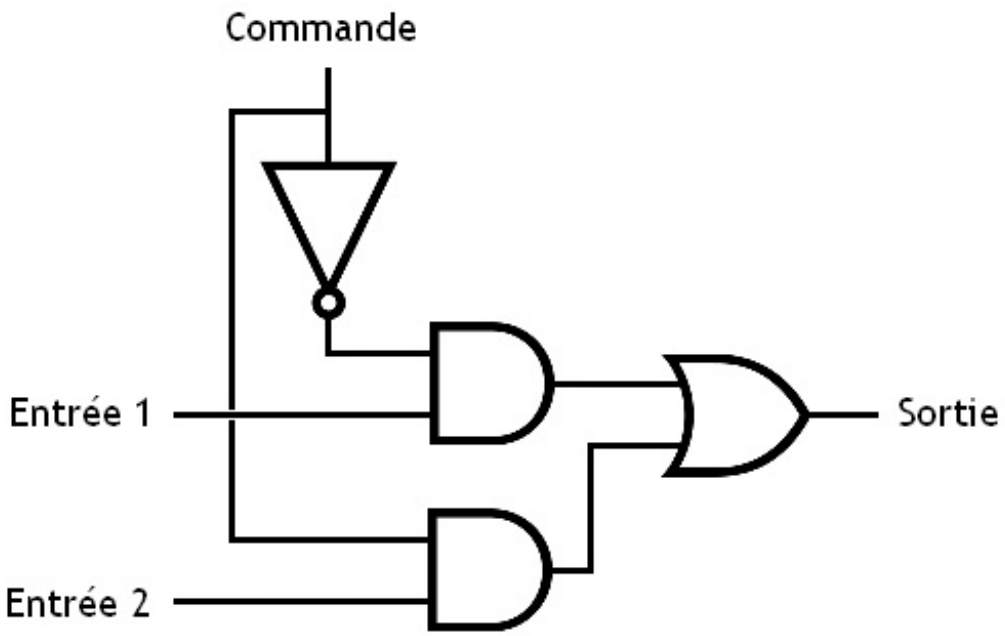


La table de vérité d'un multiplexeur est assez simple, comme vous pourrez en juger. Dans ce qui suit, on prendra nommera les deux entrées du multiplexeur E1 et E2, sa sortie S, et son entrée de commande C. La table de vérité du circuit ressemble donc à cela :

Entrée C	Entrée E1	Entrée E2	Sortie S

0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

En utilisant la méthode vue au chapitre 3, on arrive alors à trouver l'équation logique suivante : $S = (E1.\overline{C}) + (E2.C)$.
Cela nous donne donc le circuit suivant :



Sachez toutefois que les multiplexeurs utilisés dans nos ordinateurs ne sont pas forcément fabriqués avec des portes logiques. Ils sont fabriqués directement avec des transistors, afin de faire des économies.

Décaleur logique

Voyons maintenant comment créer un décaleur simple vers la droite. Ce décaleur va pouvoir décaler un nombre, vers la droite, d'un nombre de rang variable. En effet, on pourra décaler notre nombre de 2 rangs, de 3 rangs, de 4 rangs, etc.

Comment gérer ce nombre de rangs variables ?

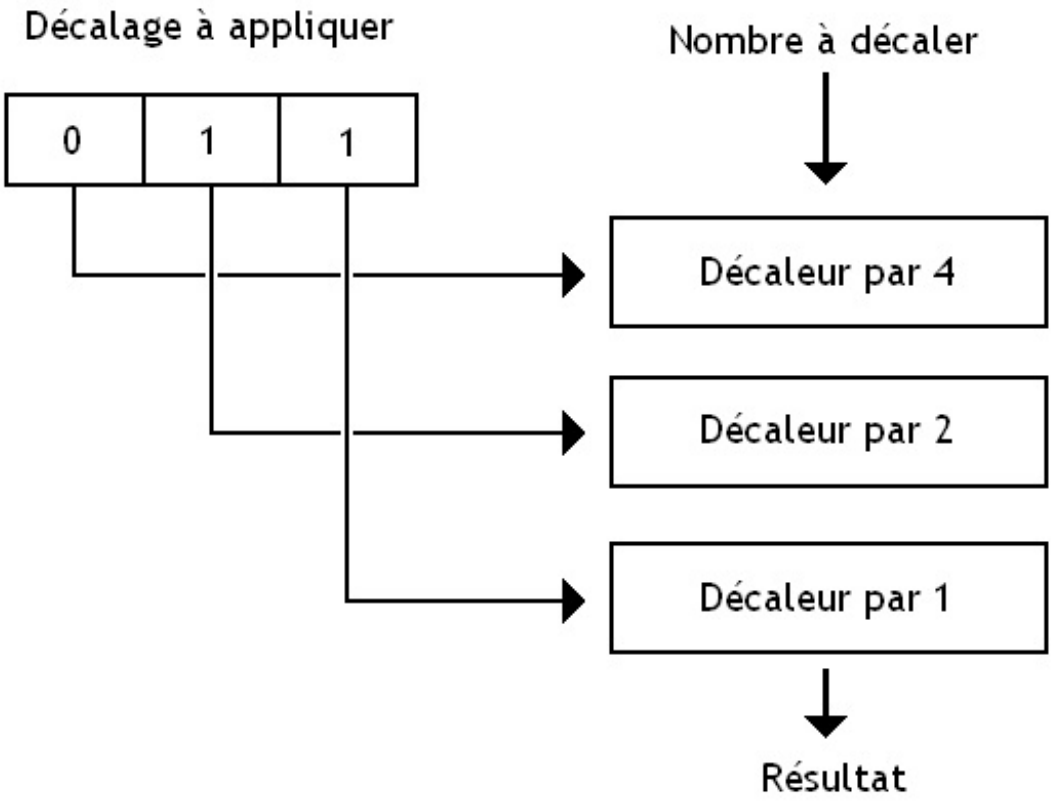
Tout d'abord, il faudra préciser ce nombre de rangs duquel on veut décaler à notre circuit. Celui-ci devra donc comporter des entrées pour spécifier de combien on veut décaler notre nombre. Reste à savoir comment créer notre circuit.

Principe

Ensuite, on peut faire une remarque simple : décaler vers la droite de 6 rangs, c'est équivalent à décaler notre nombre vers la droite de 4 rangs, et re-décaler le tout de 2 rangs. Même chose pour 7 rangs : cela consiste à décaler de 4 rangs, re-décaler de 2 rangs et enfin re-décaler d'un rang. En suivant notre idée jusqu'au bout, on se rend compte qu'on peut créer un décaleur à partir

de décaleur plus simples, reliés en cascade, qu'il suffira d'activer ou désactiver suivant la valeur du nombre de rangs qu'il faut décaler.

Le nombre de rangs par lequel on va devoir décaler est un nombre, qui est évidemment stocké en binaire dans notre ordinateur. Celui s'écrit donc sous la forme d'une somme de puissances de deux (relisez le premier chapitre si vous avez oublié). On peut donc utiliser la méthode suivante : **chaque bit de ce nombre servira à actionner le décaleur qui déplace d'un nombre de rangs égal à la valeur du bit**. Cela permet d'utiliser des déclaeurs qui décalent par 1, 2,4, 8, ou toute autre puissance de 2.



Décaleur élémentaire

Reste à savoir comment créer ces décaleurs qu'on peut activer ou désactiver à la demande. On va prendre comme exemple un décaleur par 4, pour se simplifier la vie. Mais ce que je vais dire pourra être adapté pour créer des décaleurs par 1, par 2, par 8, etc. Commençons par décrire le comportement de ce décaleur. Celui-ci prend en entrée un nombre à décaler (ici, ce sera un nombre de 8 bits qu'on nommera A). Sa sortie vaudra : soit le nombre tel qu'il est passé en entrée (le décaleur est inactif), soit le nombre décalé de 4 rangs.

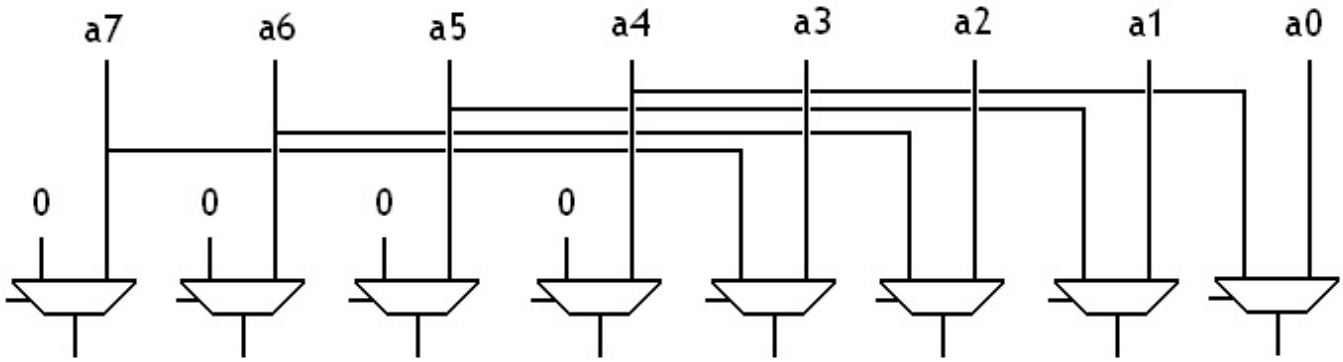
Ainsi, si je prend un nombre A, composé des bits a7, a6, a5, a4, a3, a2, a1, a0 ; (cités dans l'ordre), mon résultat sera :

- soit le nombre composé des chiffres a7, a6, a5, a4, a3, a2, a1, a0 : on n'effectue pas de décalage ;
- soit le nombre composé des chiffres 0, 0, 0, 0, a7, a6, a5, a4 : on effectue un décalage par 4.

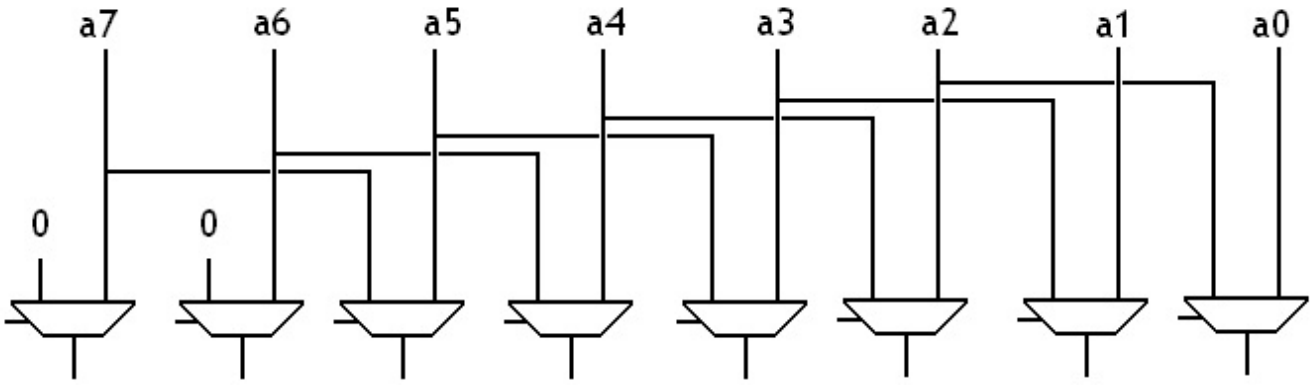
On voit donc qu'il existe deux choix possibles pour chaque bit de sortie : par exemple, le bit de poids fort peut prend deux valeurs : soit 0, soit a7. Pareil pour le 4éme bit en partant de la droite du résultat : celui-ci vaut soit a7, soit a3. On se retrouve donc avec deux choix pour chaque bit de sortie, qu'on doit sélectionner au besoin. Je ne sais pas si vous avez remarqué, mais c'est exactement ce que va faire notre multiplexeur : il va choisir deux entrées possibles et en recopier une sur sa sortie en fonction de son entrée de commande. Il nous suffira donc d'utiliser des multiplexeurs pour effectuer ce choix.

Par exemple, pour le choix du bit de poids faible du résultat, celui-ci vaut soit a7, soit 0 : il suffit d'utiliser un multiplexeur prenant le bit a7 sur son entrée 1, et un 0 sur son entrée 0. Il suffira de régler le multiplexeur pour choisir le bon bit. Il suffit de faire la même chose pour tous les autres bits, et le tour est joué. Vous devriez avoir compris le principe et êtes maintenant censés pouvoir créer un décaleur tout seul, en faisant la même chose pour les bits qui restent.

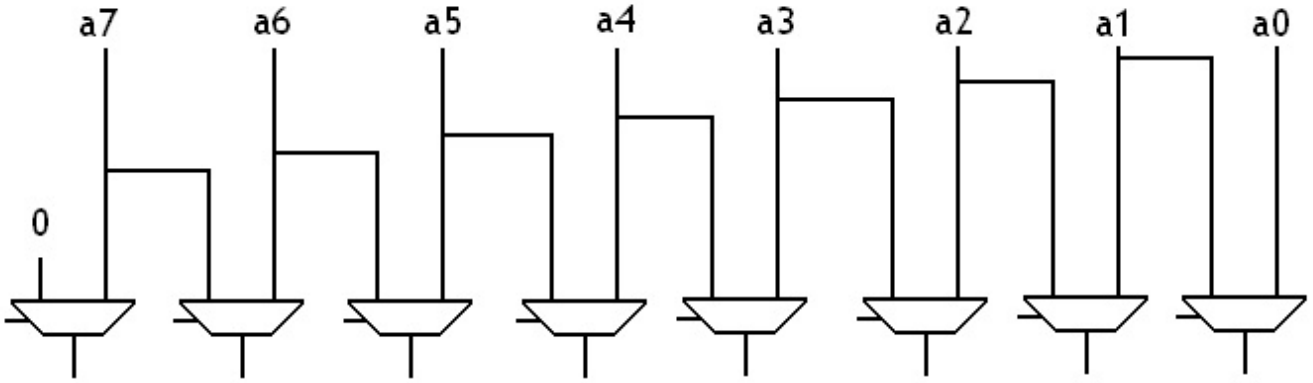
Décaleur par 4



Décaleur par 2



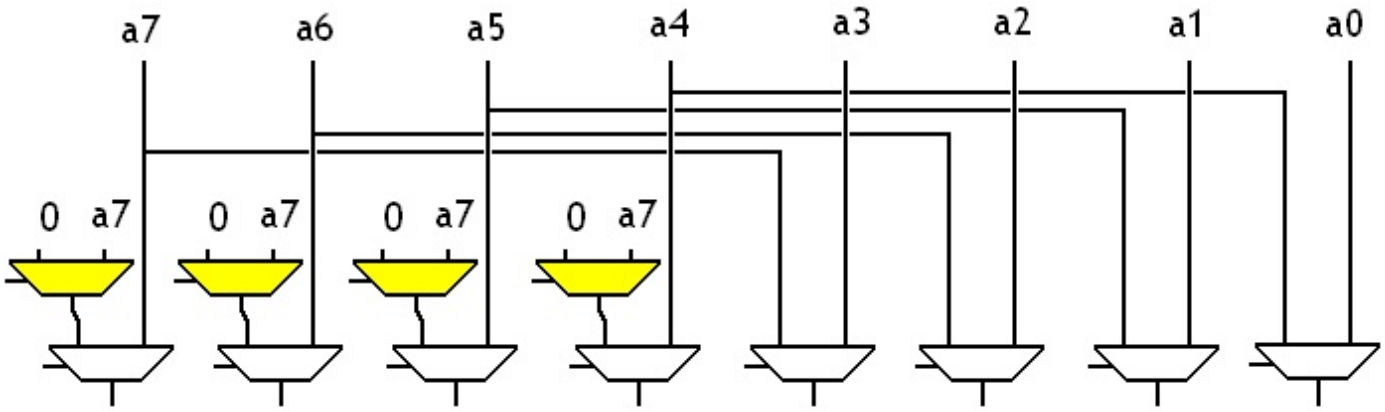
Décaleur par 1



Décaleur arithmétique

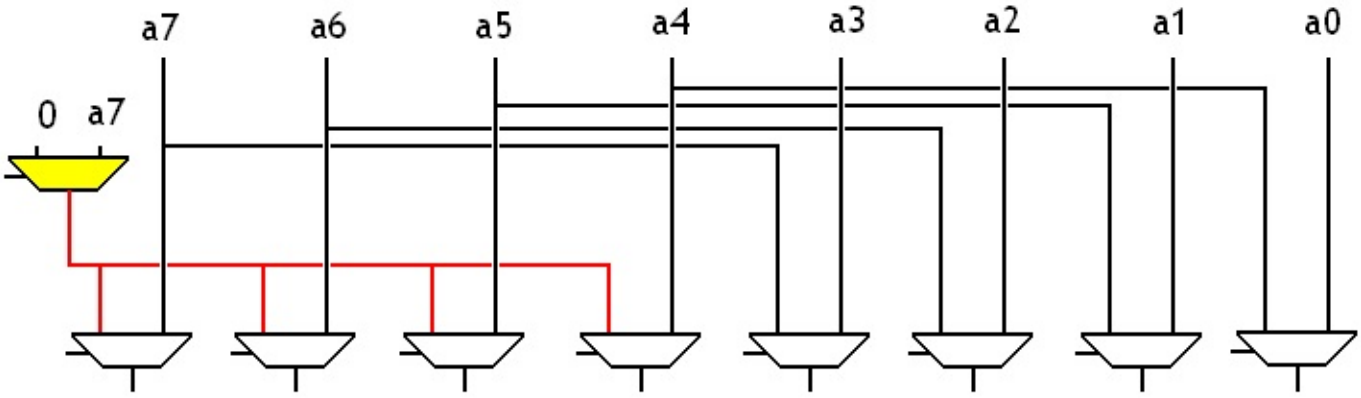
Un décaleur arithmétique peut être crée de la même façon. Plus intéressant, on peut modifier le schéma vu au-dessus pour lui permettre d'effectuer des décalages arithmétiques en plus des décalages logiques. Il suffit simplement d'ajouter un ou plusieurs multiplexeurs pour chaque décaleur élémentaire par 1, 2, 4, etc. Il suffit simplement que ce ou ces multiplexeurs choisisse quoi envoyer sur l'entrée de l'ancienne couche : soit un 0 (décalage logique), soit le bit de signe (décalage arithmétique).

Exemple avec un décaleur par 4



Ou encore avec un seul multiplexeur, mais plus de fils.

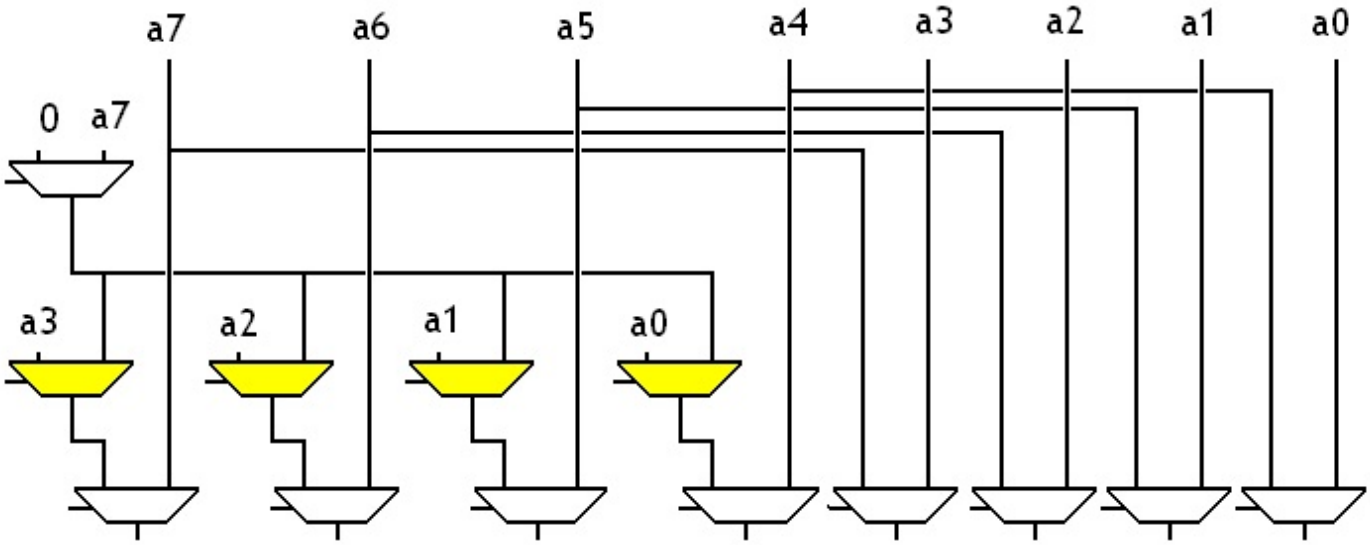
Exemple avec un décaleur par 4



Rotateur

Et ce qui peut être fait pour le décalage arithmétique peut aussi l'être pour les rotations. On peut transformer notre circuit en circuit encore plus généraliste, capable de faire des rotations en plus des décalages en rajoutant quelques multiplexeurs pour choisir les bits à envoyer sur les entrées des décaleurs.

Par exemple, on peut rajouter une couche de multiplexeurs pour faire en sorte que notre décaleurs par 4 puisse faire à la fois des décalages par 4 et des rotations par 4. Pour cela, il suffit de choisir quoi mettre sur les 4 bits de poids fort. Si c'est un décalage par 4, notre circuit devra mettre ces bits de poids fort à 0, tandis qu'il devra recopier les 4 bits de poids faible si c'est une rotation. Pour choisir entre un zéro ou le bit voulu du nombre d'entrée, il suffit de rajouter des multiplexeurs.



Bien évidemment, on peut faire la même chose pour les rotateurs par 2, 1 , etc. Et ainsi obtenir de quoi effectuer des rotations en plus des décalages.

Barell shifter

Avec tout ce qui a été dit plus haut, on est donc arrivé à créer un circuit capable d'effectuer aussi bien des rotations que des décalages : ce fameux circuit s'appelle un ***barrel shifter***, et est utilisé dans certains processeurs modernes, dans une version un peu plus améliorée. Il existe d'autres types de *Barrel shifter* qu'on a pas évoqués dans ce chapitre : ce sont les *mask barrel shifter*. Pour ceux qui sont intéressés, voici un peu de documentation sur ces décaleurs un peu spéciaux : Mask Barrel Shifters.

Addition

Voyons maintenant un circuit capable d'additionner deux nombres : l'**additionneur**. Dans la version qu'on va voir, ce circuit manipulera des nombres strictement positifs ou des nombres codés en complètement à deux, ou en complément à un.

Additionneur à propagation de retenue

Nous allons commencer par l'**additionneur à propagation de retenue**. L'idée derrière ce circuit est très simple : elle consiste à poser l'addition comme nous avons l'habitude de la faire en décimal. Tout d'abord, voyons comment additionner deux bits.
En binaire, l'addition de deux bits est très simple, jugez plutôt :

- **0 + 0 = 0** ;
- **0 + 1 = 1** ;
- **1 + 0 = 1** ;
- **1 + 1 = 10**, ce qui est équivalent à 0 plus une retenue.

On voit que l'addition de deux bits n'est pas forcément codée sur un seul bits : on peut avoir une retenue.

Pour effectuer une addition, on va additionner deux bits de même rang (mieux dit : de même poids) en tenant compte d'une éventuelle retenue. Évidemment, on commence par les bits les plus à droite, comme en décimal.

Par exemple :

0 1 0 0 1 0
+ 0 1 0 1 0

= 1 1 1 0 0

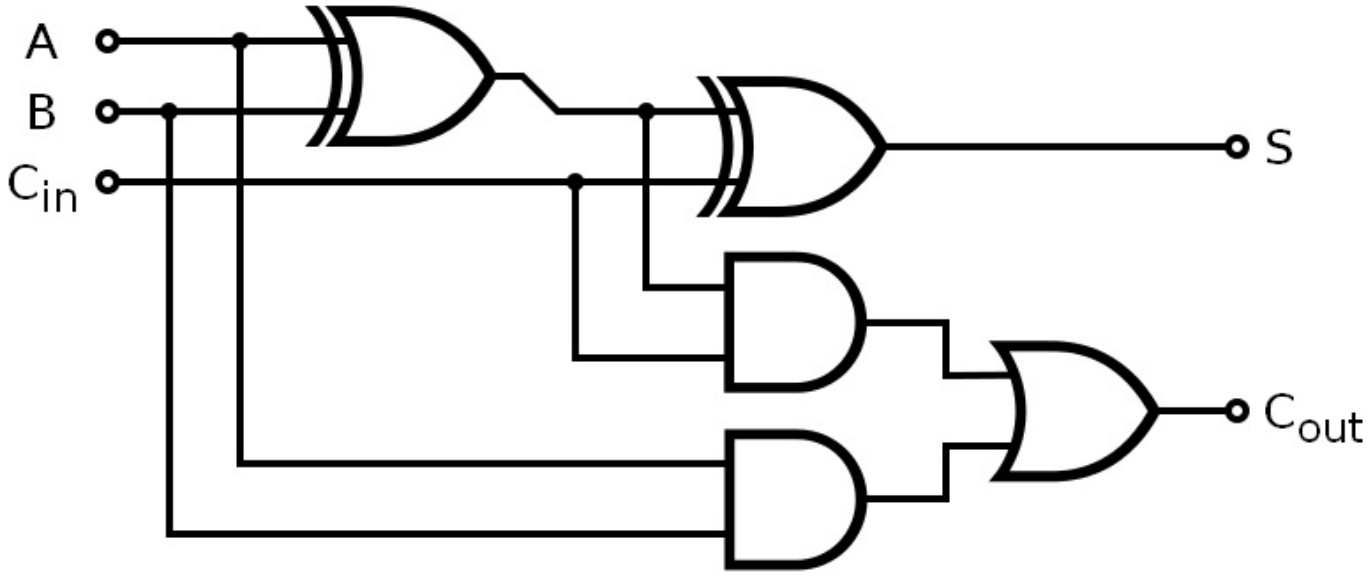
Additionneur complet

Pour effectuer notre addition, tout repose sur l'addition de deux bits, et d'une éventuelle retenue : on va devoir créer un circuit pour. Notre circuit possédera deux sorties : une pour le résultat, et une pour la retenue. En plus de pouvoir additionner deux bits, il faut prendre en compte la retenue de l'addition des bits précédents, qui viendra s'ajouter à nos deux bits.

Pour cela, on va créer un circuit capable d'additionner trois bits qu'on appellera : l'**additionneur complet**. Ce circuit comprendra trois entrées : les deux bits a et b à additionner, et une entrée Cin, pour la retenue de l'addition des bits précédents. Il aura aussi deux sorties : une pour la retenue du résultat, qu'on nommera Cout, et une autre pour le résultat de l'addition, qu'on nommera Sum.

Bit a	Bit b	Cin		Sum	Cout
0	0	0		0	0
0	0	1		1	0
0	1	0		1	0
0	1	1		0	1
1	0	0		1	0
1	0	1		0	1
1	1	0		0	1
1	1	1		1	1

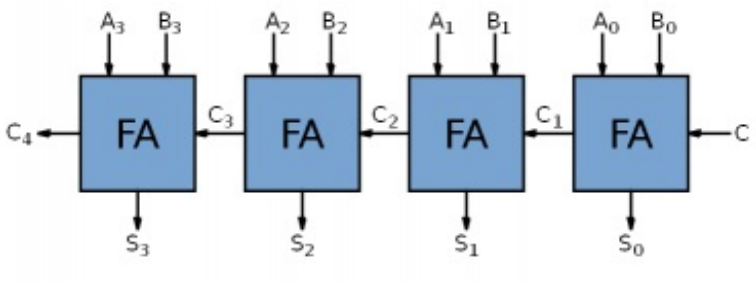
En utilisant les techniques vues au chapitre 3 de ce tutoriel, on peut alors trouver que le câblage de notre additionneur complet.



Il existe de nombreuses façons d'implémenter un additionneur complet. On peut parfaitement créer un additionneur complet sans utiliser de portes logiques, mais en travaillant directement avec des transistors : cela permet d'avoir quelques opportunités d'améliorations assez sympathiques. On peut ainsi créer des additionneurs complets comprenant bien moins de transistors que celui vu au-dessus.

Circuit complet

Maintenant, on a tout ce qu'il faut pour créer ce qu'on appelle un additionneur à propagation de retenue. Il suffit de cabler nos additionneurs les uns à la suite des autres. Par exemple, pour additionner deux nombres de 4 bits, on obtiendra le circuit suivant.



Notez la présence de l'entrée de retenue C. Presque tous les additionneurs de notre ordinateur on une entrée de retenue comme celle-ci, afin de faciliter l'implémentation de certaines opérations comme l'inversion de signe, l'incrémentation, etc.

Performances

Pour votre information, ce circuit a un gros problème : chaque additionneur doit attendre que la retenue de l'addition précédente soit disponible pour donner son résultat. Pour obtenir le résultat, Les retenues doivent se propager à travers le circuit, du premier additionneur jusqu'au dernier. Et ça prend du temps, ce qui fait que ce circuit naïf ne convient pas pour des processeurs destinés à être rapides.

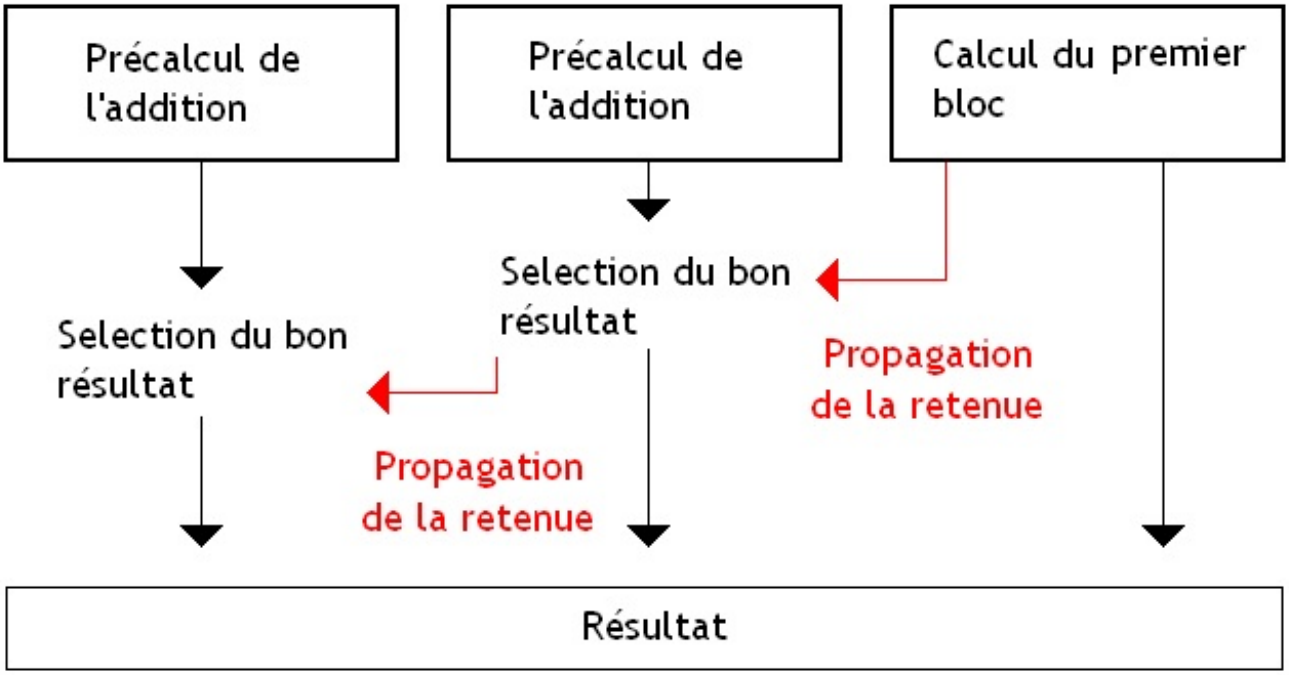
Or, l'addition est une opération très fréquente dans nos programmes. De plus, ces additionneurs sont utilisés dans d'autres circuits, pour calculer d'autres opérations arithmétiques, comme les multiplications, les soustractions, etc. La rapidité de ces opérations, dont certaines sont très complexes, dépend fortement de la rapidité des additionneurs qu'elles vont utiliser. Il nous faut donc créer des additionneurs un peu plus rapides.

L'additionneur à sélection de retenue

Pour cela, il existe une solution assez simple qui consiste à casser notre additionneur à propagation de retenue en plusieurs petits additionneurs qu'on organise différemment. Un additionneur conçu ainsi s'appelle un **additionneur à sélection de retenue**.

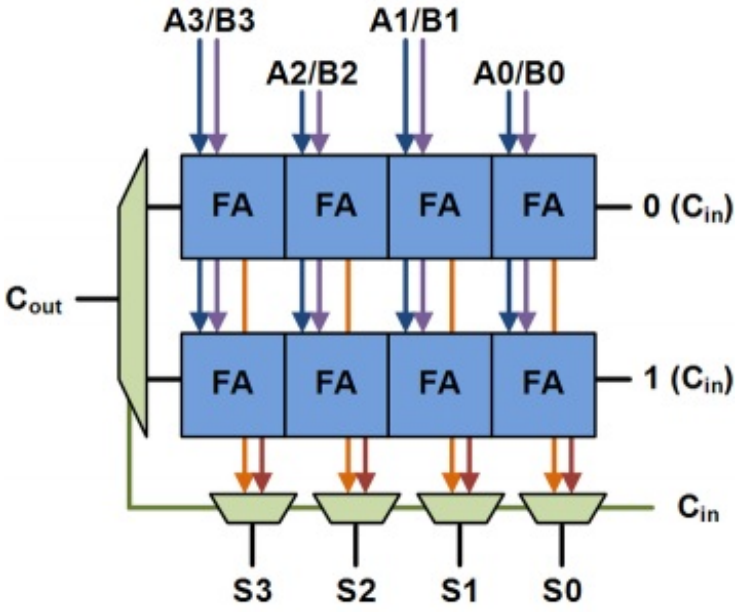
Principe

Cet additionneur va découper nos deux nombres à additionner en blocs, qui se feront additionner en deux versions : une avec la retenue du bloc précédent valant zéro, et une autre version avec la retenue du bloc précédent valant 1. Il suffira alors de choisir le bon résultat une fois cette retenue connue. On gagne ainsi du temps en calculant à l'avance les valeurs de certains bits du résultat, sans connaitre la valeur de la retenue.



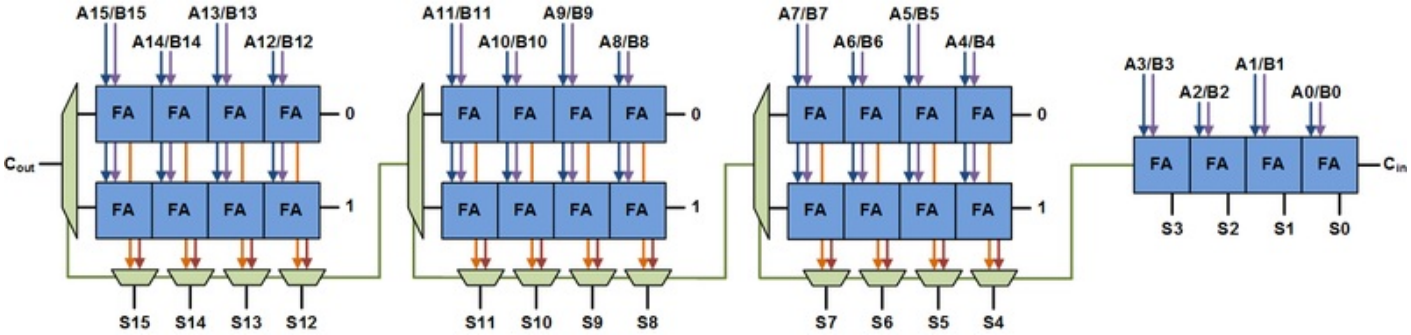
Bloc de base

Un tel additionneur à sélection de retenue est composé de briques de base, capables de pré-calculer un morceau du résultat de l'addition et de choisir le bon résultat. Chacune de ces briques de base sera composé de deux additionneurs : bl'un calculant la somme des bits (retenue incluse) passés en entrée si l'entrée de retenue est à zéro ; et l'autre faisant la même chose mais avec l'entrée de retenue à 1. La sélection du bon résultat se fait en fonction de l'entrée de retenue : il suffit de relier l'entrée de retenue sur l'entrée de commande d'un multiplexeur.



Circuit

En faisant ainsi, il suffira juste de relier les entrées de retenues d'un bloc de base aux sorties de retenues du bloc précédents.



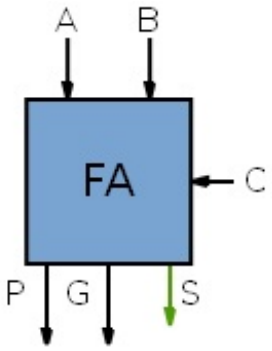
Petit détail : sur certains additionneurs à sélection de retenue, les blocs de base n'ont pas la même taille. Cela permet de tenir compte des temps de propagation des retenues entre les blocs.

Additionneurs à anticipation de retenue

D'autres additionneurs encore plus rapides existent. Certains de ces additionneurs (la majorité) utilisent pour cela une astuce très simple : au lieu de calculer les retenues unes par unes, ils calculent toutes les retenues en parallèle à partir de la valeur de tout ou partie des bits précédents. On les appelle des **additionneurs à anticipation de retenue**. Ces additionneurs sont conçus avec quelques principes simples en tête.

Additionneurs

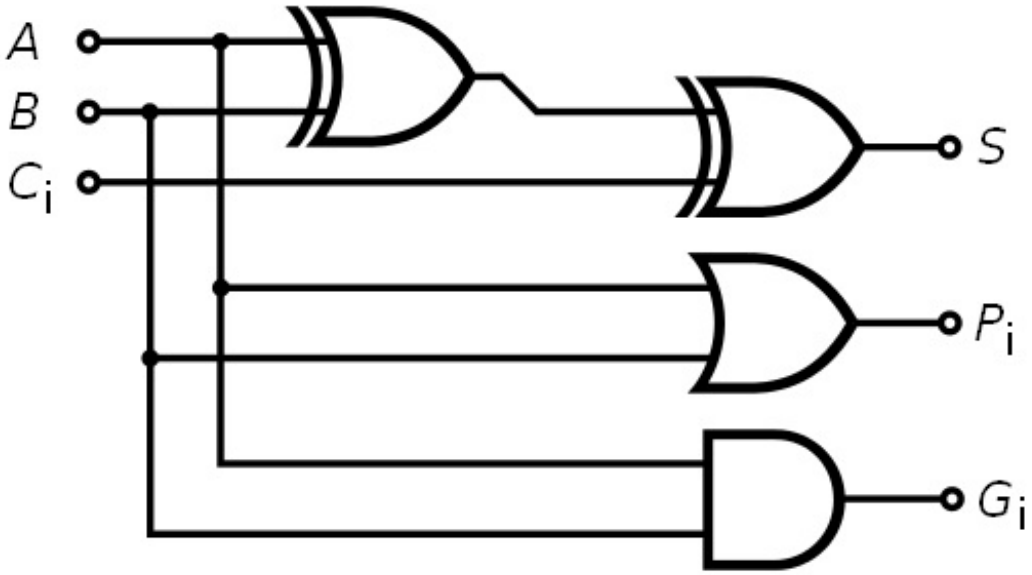
Ceux-ci utilisent les concepts de génération et de propagation de retenue. Leurs additionneurs complets sont légèrement modifiés, et possèdent deux sorties qui remplacent la sortie de retenue. Ces deux entrées vont servir à indiquer si notre additionneur complet va générer ou propager une retenue.



Le fait que notre additionneur génère une retenue sera indiqué par sa sortie G. Un additionneur complet va générer une retenue si on se retrouve avec un 1 sur la sortie de retenue, quelque soit la retenue envoyée en entrée. Pour cela, il faut que les deux bits qu'il additionne soient à 1. La valeur à mettre sur cette sortie est donc un simple ET entre les deux bits à additionner.

Le fait que notre additionneur propage une retenue sera indiqué par sa sortie P. Un additionneur complet va propager une retenue si la retenue en sortie vaut 1 si et seulement si la retenue placée en entrée vaut 1. Cela est possible si un des deux bits placé en entrée vaut 1. La valeur à mettre sur cette sortie est donc un simple OU entre les deux bits à additionner.

Notre additionneur ressemble donc à ceci :



Anticipation de retenue

La retenue finale d'un additionneur complet est égale à 1 si celui-ci génère une retenue ou s'il en propage une. Il s'agit donc d'un simple OU entre les sorties P et G. Ainsi, l'addition des bits de rangs **i** va produire une retenue **Ci**, qui est égale à **Gi + (Pi . Ci-1)**. L'astuce des additionneurs à anticipation de retenue consiste à remplacer le terme **Ci-1** par sa valeur calculée avant.

Par exemple, je prends un additionneur 4 bits. Je dispose de deux nombres A et B, contenant chacun 4 bits : A3, A2, A1, et A0 pour le nombre A, et B3, B2, B1, et B0 pour le nombre B. Si j'effectue les remplacements, j'obtiens les formules suivantes :

- C1 = G0 + (P0 . C0) ;
- C2 = G1 + (P1 . G0) + (P1 . P0 . C0) ;
- C3 = G2 + (P2 . G1) + (P2 . P1 . G0) + (P2 . P1 . P0 . C0) ;
- C4 = G3 + (P3 . G2) + (P3 . P2 . G1) + (P3 . P2 . P1 . G0) + (P3 . P2 . P1 . P0 . C0) ;

Ces formules nous permettent de déduire la valeur d'une retenue directement à partir des sorties de nos additionneurs. On effectue les calculs de nos sommes et des bits P et G pour chaque additionneur en parallèle, et on en déduit directement les retenues sans devoir les propager.

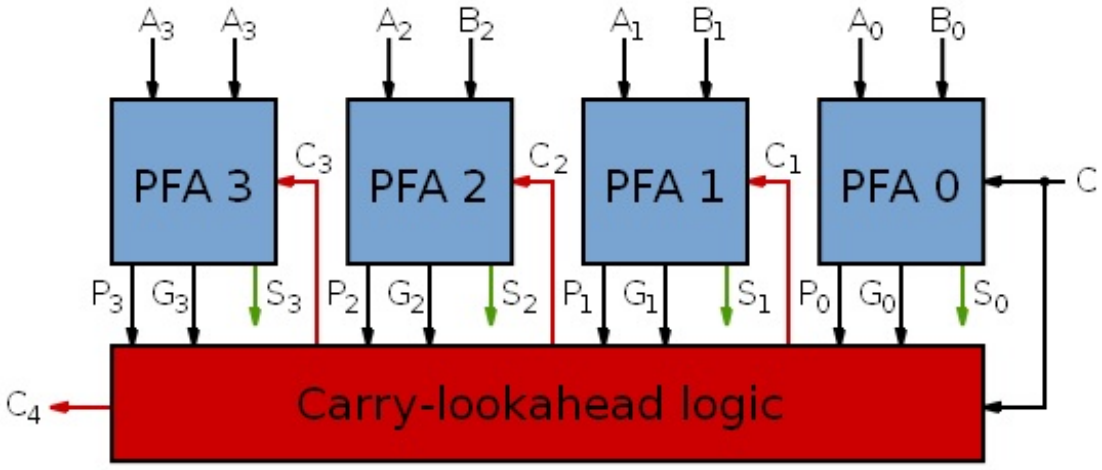
Bien sûr, il faut un certain temps pour déduire la retenue en fonction des bits P et G adéquats. Mais ce temps est nettement inférieur au temps qui serait mit pour propager une retenue avec un additionneur à propagation de retenue. Plus précisément, ce temps de propagation des retenues est proportionnel au nombre de bits des nombres à additionner. Pour un additionneur à

sélection de retenue, on est proche de la racine carrée du nombre de bits.

En comparaison, le temps mit pour anticiper les retenues est égal au logarithme du nombre de bits. Si vous ne savez pas ce qu'est un logarithme ou que vous avez du mal avec les maths, sachez juste que c'est beaucoup plus rapide, surtout quand le nombre de bits augmente.

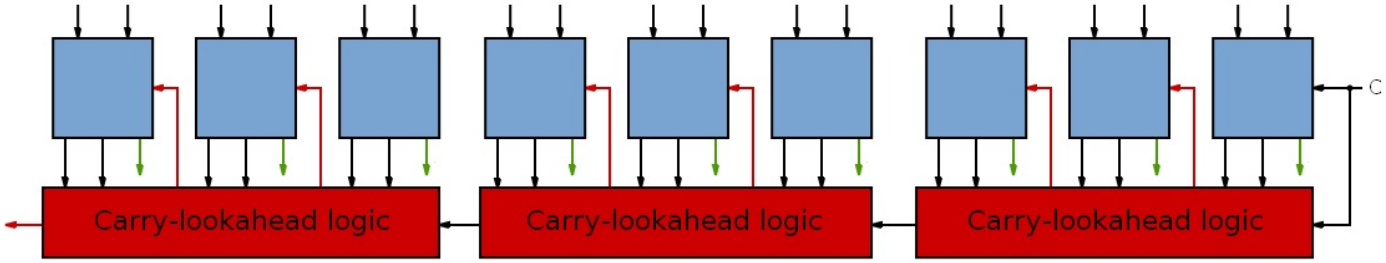
Additionneur

Notre additionneur à anticipation de retenue est donc composé d'une couche d'additionneurs, et d'un paquet de portes logiques qui permettent de déduire les retenues de façon anticipée. Ce paquet de portes logiques est souvent rassemblé dans une unité spéciale, l'unité d'anticipation de retenue, aussi appelée ***Carry Lookahead Unit***.



Améliorations

Ceci dit, utiliser un additionneur à anticipation de retenue sur des nombres très grands (16/32bits) serait du suicide : cela utiliserait trop de portes logiques, et poserait quelques problèmes techniques assez difficiles à résoudre. Pour éviter tout problème, nos additionneurs à anticipation de retenue sont souvent découpés en blocs, capables d'additionner N bits. Suivant l'additionneur, on peut avoir une anticipation de retenue entre les blocs et une propagation de retenue dans les blocs, ou l'inverse.

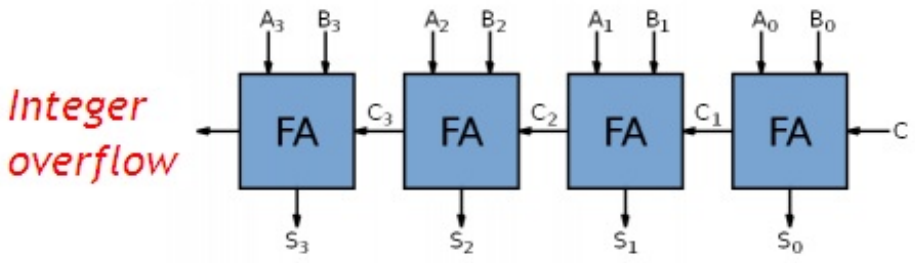


Les Overflows

Les instructions arithmétiques et quelques autres manipulent des entiers de taille fixe, qui ne peuvent prendre leurs valeurs que dans un intervalle déterminé par une valeur minimale et une valeur maximale. Si le résultat d'un calcul sur des nombres entiers sort de cet intervalle, il ne pas être représenté dans notre ordinateur : il se produit ce qu'on appelle un ***integer overflow***. Et quand un *integer overflow* a eu lieu, il vaut mieux prévenir ! Sur certains processeurs, on détecte ces *integer overflow* de façon logicielle, ou en utilisant des instructions spécialisées. Mais sur certains processeurs, cette détection se fait automatiquement lors de l'addition. Pire : ils peuvent parfois corriger ces *integer overflow* automatiquement. Dans ce qui va suivre, on voir comment. Malheureusement, la gestion des *integer overflow* dépend de la représentation des nombres utilisée.

Entiers strictement positifs, non signés

Commençons par étudier la gestion des *integer overflow* pour les entiers non-signés. Pour détecter cet *integer overflow*, on va devoir rajouter une sortie supplémentaire à notre additionneur. Cette sortie sera positionnée à 1 si un *integer overflow* a lieu. Bien sûr, il faudra rajouter un peu de circuiterie pour détecter cet *integer overflow*. Détecter un *integer overflow* avec les additionneurs vus au-dessus est super-simple : il suffit simplement de regarder la dernière sortie de retenue. Dans ce qui va suivre, je vais utiliser un additionneur à propagation de retenue pour les exemples, mais le principe est strictement le même pour les autres additionneurs.



Une fois détecté, cet *integer overflow* peut être géré par le processeur ou par le logiciel.

Gestion logicielle

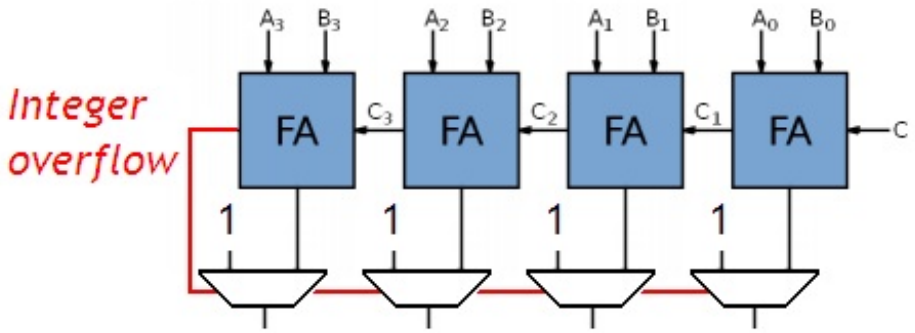
Si il est géré par le logiciel, celui-ci peut décider de passer outre, ou de le corriger. Encore faut-il qu'il sache qu'il y a eu un *integer overflow*. Dans la plupart des cas, un bit du registre d'état est dédié à cette gestion logicielle des *integer overflow*. Ce bit est mit automatiquement à 1 en cas d' *integer overflow*. Un programme qui veut gérer cet *integer overflow* a juste à utiliser un branchement conditionnel qui agira en fonction de la valeur de ce bit. Ce branchement renverra le processeur vers un sous-programme de gestion d'*integer overflow*. Ce bit est relié directement sur la sortie de l'additionneur qui indique l'occurrence d'un *integer overflow*.

Si l'*Overflow* n'est pas corrigé automatiquement par le processeur, celui-ci ne conserve que les bits de poids faibles du résultat : les bits en trop sont simplement ignorés. Le problème avec ce genre d'arithmétique, c'est qu'une opération entre deux grand nombres peut donner un résultat très petit. Par exemple, si je dispose de registres 4 bits et que je souhaite faire l'addition $1111 + 0010$ (ce qui donne 15 + 2), le résultat est censé être 10001 (17), ce qui est un résultat plus grand que la taille d'un registre. En conservant les 4 bits de poids faible, j'obtiens 0001 (1). En clair, un résultat très grand est transformé en un résultat très petit. Si vous regardez bien, les circuits vus au-dessus sont dans ce cas.

Gestion matérielle

D'autres processeurs utilisent ce qu'on appelle l'**Arithmétique_saturée** : si un résultat est trop grand au point de générer un *integer overflow*, on arrondi le résultat au plus grand entier supporté par le processeur. Les processeurs qui utilisent l'arithmétique saturée sont souvent des DSP, qui doivent manipuler du signal ou de la vidéo. Certaines instructions de nos processeurs x86 (certaines instructions SSE) font leurs calculs en arithmétique saturée.

Par contre, les circuits capables de calculer en arithmétique saturée sont un peu tout petit peu plus complexes que leurs collègues qui ne travaillent pas en arithmétique saturée. Il est toutefois assez simple de modifier nos additionneurs du dessus pour qu'ils fonctionnent en arithmétique saturée. Il suffit pour cela de rajouter une couche de multiplexeurs, qui enverra sur sa sortie :soit le résultat de l'addition, soit le plus grand nombre entier géré par le processeur. Cette couche de multiplexeurs est commandée par le signal d'*Overflow*, disponible en sortie de notre additionneur.



Complément à deux et complément à un

Pour les nombres codés en complément à deux, la situation se corse. Si vous vous rappelez le chapitre 1, j'ai clairement dit que les calculs sur des nombres en complètement à deux utilisent les règles de l'arithmétique modulaire : ces calculs seront faits sur des entiers ayant un nombre de bits fixé une fois pour toute. Si un résultat dépasse ce nombre de bits fixé, on ne conserve pas les bits en trop. C'est une condition nécessaire pour pouvoir faire nos calculs. A priori, on peut donc penser que dans ces conditions, les *integer overflow* sont une chose parfaitement normale, qui nous permet d'avoir des résultats corrects.

Néanmoins, il faut se méfier de nos intuitions : certains *integer overflow* peuvent arriver et produire des bugs assez ennuyeux.

Détection des Overflows

Si l'on tient en compte les règles du complément à deux, on sait que le bit de poids fort (le plus à gauche) permet de déterminer si le nombre est positif ou négatif : ce bit ne sert pas vraiment à représenter une valeur, mais indique le signe du nombre. Tout se passe comme si les entiers en complément à deux étaient codés sur un bit de moins, et avaient leur longueur amputé du bit de poids fort. Si le résultat d'un calcul a besoin d'un bit de plus que cette longueur, amputée du bit de poids fort), ce bit de poids fort sera écrasé, et on se retrouvera avec un *integer overflow* digne de ce nom. Par exemple, si l'on additionne les nombres **0111 1111** et **0000 0001**, le résultat sera le nombres **1000 0000**, qui est négatif ! Il y a bien eu *integer overflow* : le bit de signe aura été écrasé par un bit du résultat.

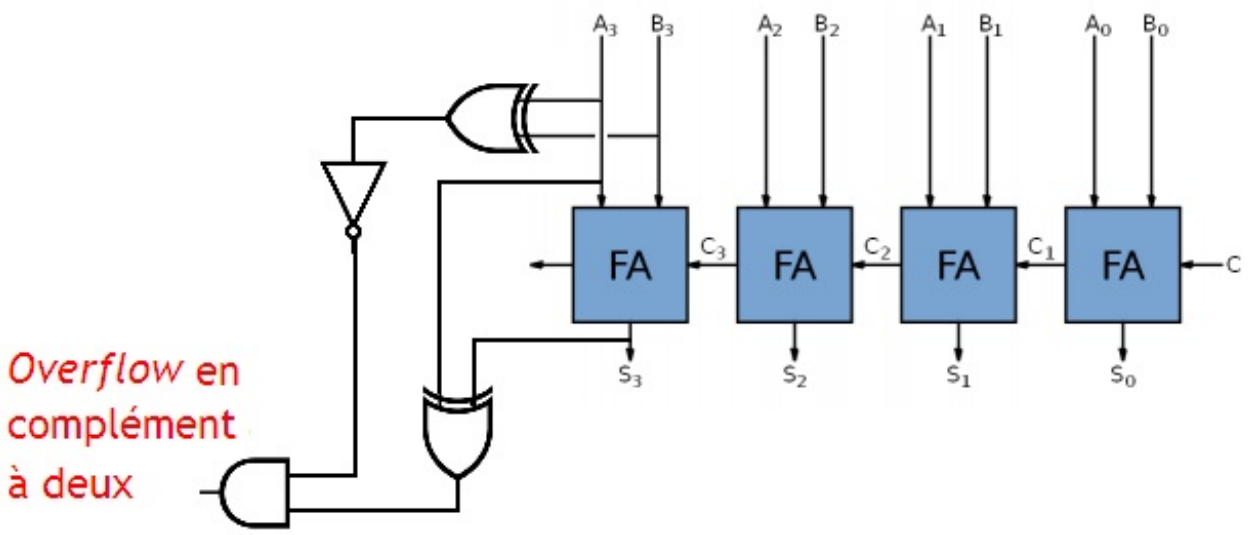
Il existe une règle simple qui permet de détecter ces *integer overflow*. L'addition (ou la multiplication) de deux nombres positifs ne peut pas être un nombre négatif : on additionne deux nombres dont le bit de signe est à 0 et que le bit de signe du résultat est à 1, on est certain d'être en face d'un *integer overflow*. Même chose pour deux nombres négatif : le résultat de l'addition ne peut pas être positif. On peut résumer cela en une phrase.

> Si deux nombres de même signe sont ajoutés, un *integer overflow* a lieu quand le bit du signe du résultat a le signe opposé.

On peut préciser que cette règle s'applique aussi pour les nombres codés en complément à 1, pour les mêmes raisons que pour le codage en complément à deux. Cette règle est aussi valable pour d'autres opérations, comme les multiplications.

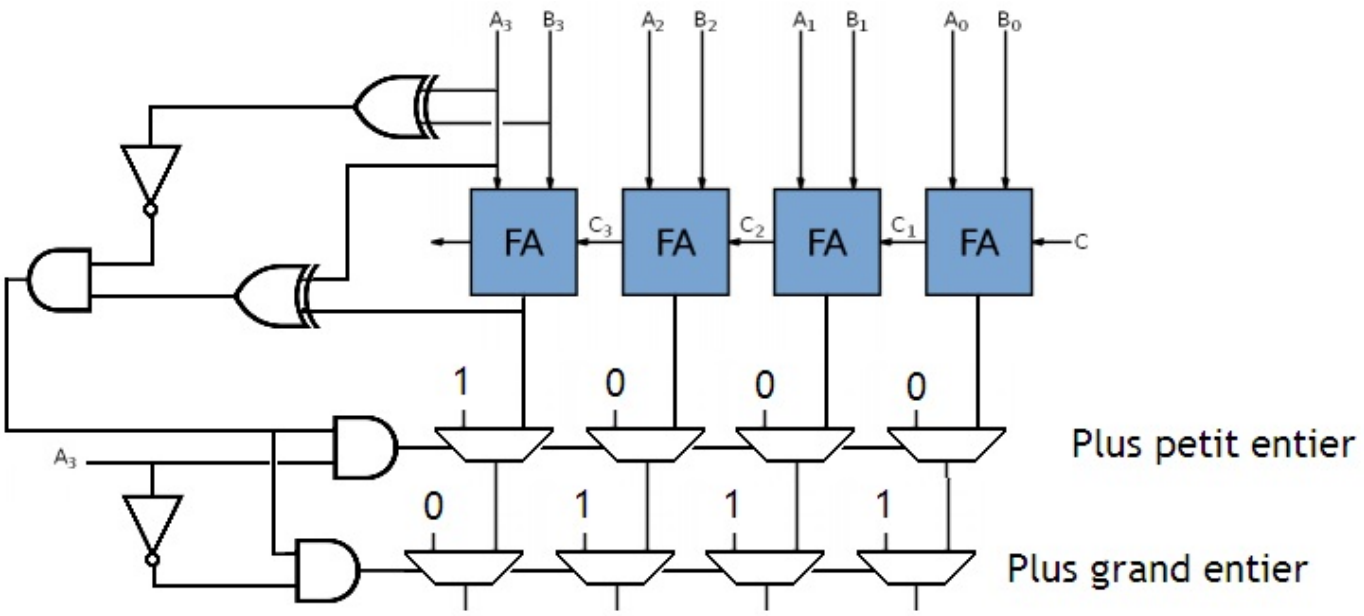
Circuit

Modifier les circuits d'au-dessus pour qu'ils détectent les *Overflows* en complément à deux est simple comme bonjour : il suffit créer un petit circuit combinatoire qui prenne en entrée les bits de signe des opérandes et du résultat, et qui fasse le calcul de l'indicateur d'*Overflow*. Voici ce que cela donne.



Correction

Encore une fois, corriger cet *Overflow* peut se faire logiciellement, ou en utilisant de l'arithmétique saturée. Mais il y a une petite subtilité avec l'arithmétique saturée : quelle est la valeur à envoyer sur la sortie ? Avec les entiers positifs, le choix était simple : il suffisait d'envoyer le plus grand entier possible. Mais en complément à deux, il faut tenir compte de deux possibilités : celle où les deux opérandes sont positives, et celle où les deux sont négatives. Dans le premier cas, on doit renvoyer le plus grand entier, et le plus petit dans le cas contraire. On a donc besoin d'une seconde couche de multiplexeurs, et on rajouter des portes pour activer chaque couche dans les bonnes circonstances.



Soustraction

On sait maintenant effectuer une addition. C'est pas mal, mais pas question de s'arrêter en chemin. Si on sait câbler une addition, câbler une soustraction n'est pas très compliqué. On va commencer par un circuit capable de soustraire deux nombres représentés en complément à deux ou en complément à un. La raison : se faciliter la vie, vu que travailler avec des entiers représentés en signe-valeur absolue est souvent plus compliqué.

Complément à deux et complément à un

Pour comprendre l'algorithme utilisé pour soustraire deux nombres représentés en complètement à deux, il va falloir faire un tout petit peu d'arithmétique élémentaire. Vous savez surement que $a - b$ et $a + (-b)$ sont deux expressions équivalentes. Sauf que si l'on regarde bien, la première expression est une soustraction, tandis que la seconde est une addition. Pour le moment, vous ne savez pas faire $a - b$ car on n'a pas encore câblé de circuit capable de faire une soustraction. Mais $a + (-b)$, vous en êtes capables : il s'agit d'une addition, qui peut être effectuée grâce au circuit vu au-dessus.

Il ne nous reste plus qu'à trouver un moyen de calculer l'opposé de b, et on pourra réutiliser l'additionneur vu précédemment pour notre calcul. Et c'est là que l'on se rend compte qu'on peut utiliser les propriétés de la représentation en complément à deux. Si vous vous souvenez du premier chapitre, j'avais dit qu'on pouvait trouver l'inverse d'un nombre positif en inversant tous les bits du nombre et en ajoutant 1. Et bien cette méthode marche aussi pour les entiers négatifs : on calcule l'inverse d'un nombre en additionnant 1 à son complément à 1.

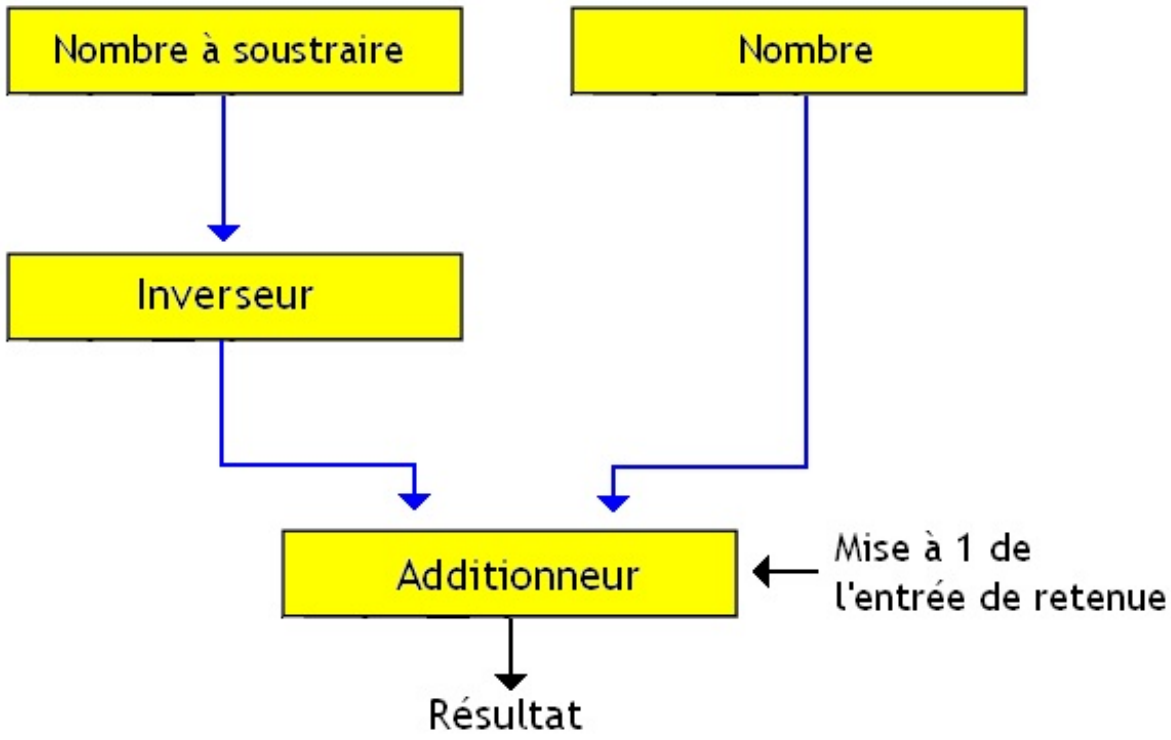
Soustraction

Notre circuit doit donc

- inverser tous les bits du nombre à soustraire ;
- ajouter 1 au résultat de cette inversion ;
- ajouter la seconde opérande au résultat calculé à l'étape 2 (l'autre nombre : celui auquel on soustrait) au résultat.

le circuit capable d'inverser tous les bits d'un nombre est évident : il s'agit d'un circuit composé uniquement de portes ***NON***, chacune d'entre elle étant reliée à un bit du nombre à inverser. Il ne nous reste plus qu'a additionner la première opérande (le nombre auquel on soustrait), augmentée de 1.

Naïvement, on pourrait se dire qu'il faudrait utiliser deux additionneurs à propagation de retenue, ou un additionneur suivi d'un circuit capable d'incrémenter (augmenter de 1) cette opérande. Mais il y a moyen de faire nettement mieux en rusant juste un chouïa. La majorité des additionneurs possède une entrée de retenue (la fameuse entrée de retenu C que je vous ais fait remarquer plus haut), pour simplifier la conception de certaines instructions. Pour additionner la seconde opérande augmentée de 1, il suffira de positionner ce bit de retenue à 1 et envoyer les opérandes sur les entrées de notre additionneur.

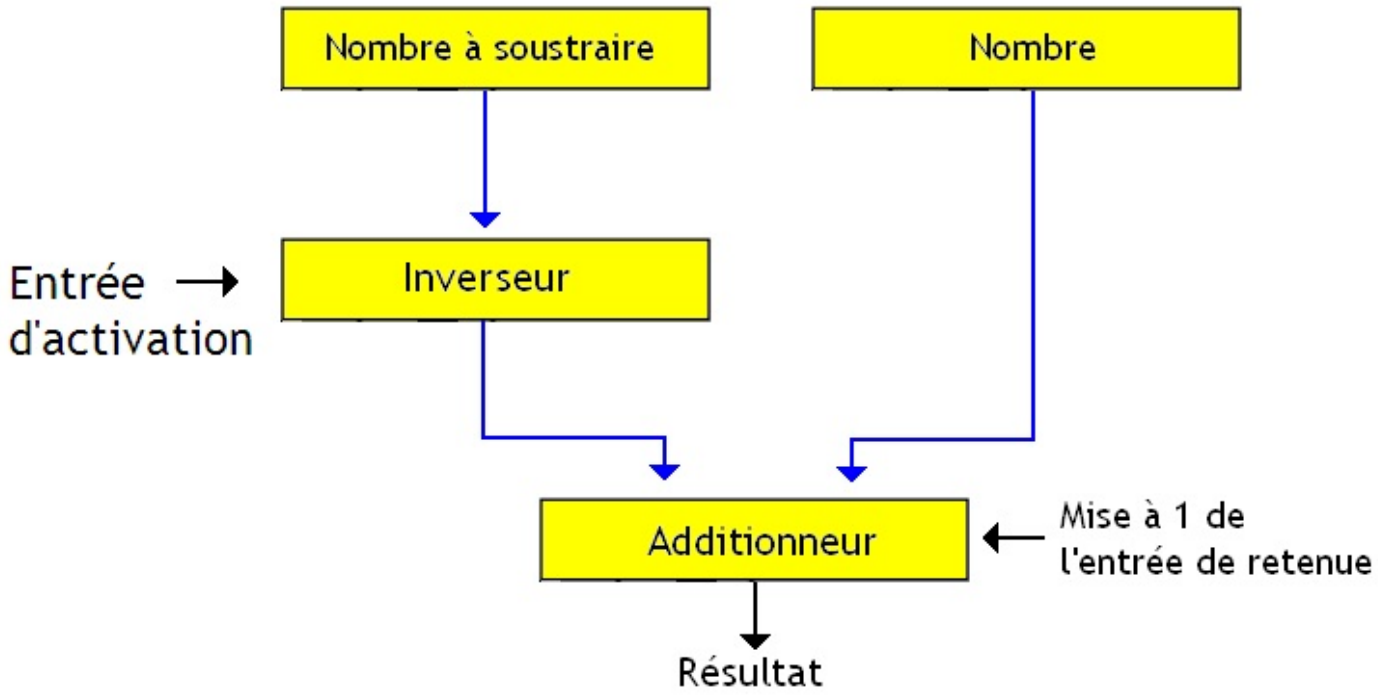


Notre circuit soustracteur est donc très simple : il est constitué du circuit inverseur vu au-dessus, auquel on relie une des deux entrées d'un additionneur sur sa sortie. Il faut juste faire en sorte de positionner la retenue de l'additionneur à 1 pour que tout fonctionne. Au fait, le circuit utilisé pour soustraire deux nombres représentés en complément à un est identique à part un détail : il n'y a pas besoin de positionner l'entrée de retenue de notre additionneur à 1 et on doit laisser celle-ci à zéro.

Addition et soustraction

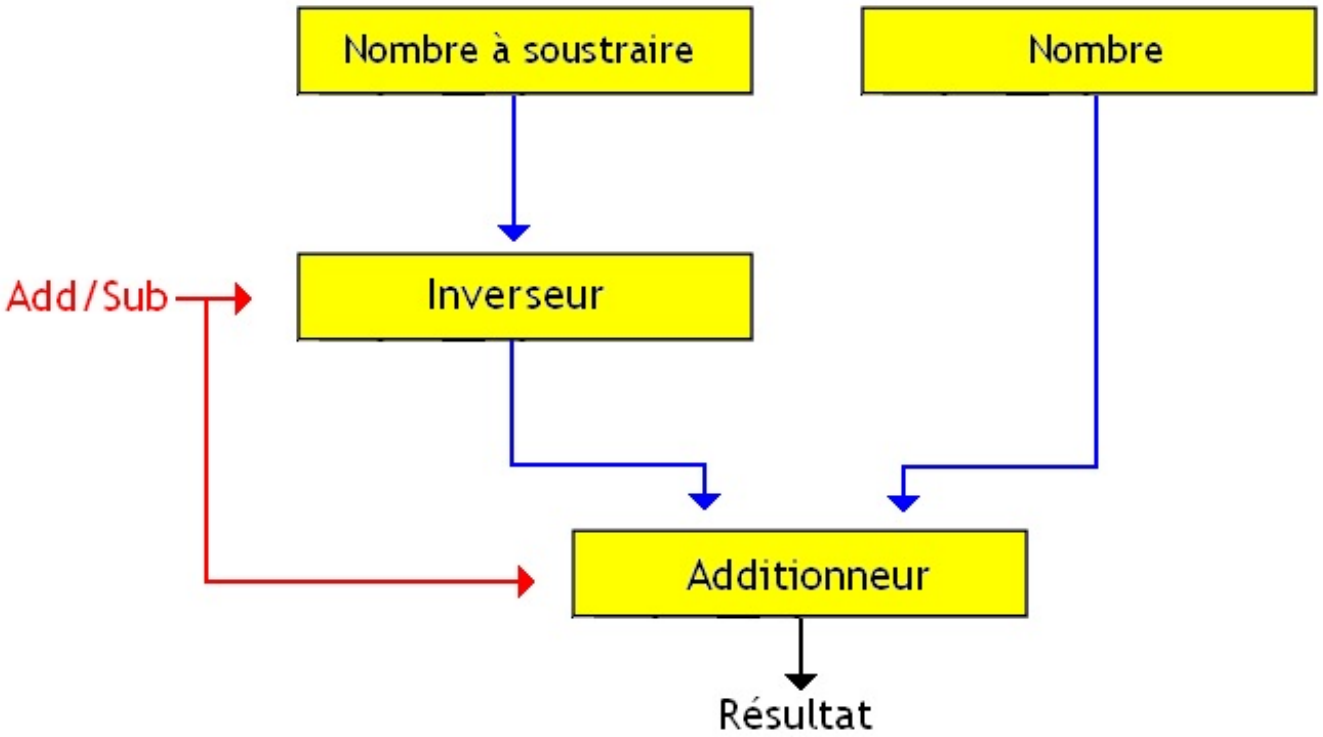
Comme je l'ai dit dans le chapitre précédent, le circuit chargé de la soustraction et celui dédié à l'addition peuvent être fusionnés dans un seul et unique circuit, capable de faire les deux. La raison est simple : l'additionneur est présent à la fois dans le circuit dédié à l'addition (normal..), et dans celui dédié aux soustractions. Il est donc possible de créer un circuit capable d'effectuer soit une addition, soit une soustraction. L'opération effectuer est choisit par un bit placé sur une entrée supplémentaire. Reste à savoir comment créer ce circuit.

La seule différence entre le circuit chargé de l'addition et celui de la soustraction tient dans l'inverseur, ainsi que dans la retenue placée sur l'entrée de l'additionneur. Pour créer notre circuit, on va donc faire en sorte que notre inverseur puisse être désactivé ou court-circuité, afin de laisser l'additionneur tranquille. Une solution consiste à créer un inverseur spécial, sur lequel on rajoute une entrée d'activation. Si cette entrée vaut 1, l'inverseur inversera l'opérande qui lui fournie en entrée. Dans le cas contraire, cet inverseur ne fera rien et recopiera l'opérande passée en entrée sur sa sortie.



Pour cela, rien de plus simple : il suffit de remplacer chaque porte ***NON*** de l'inverseur par une porte ***XOR***. Une entrée de cette porte ***XOR*** doit être reliée à un bit de l'opérande, et l'autre sera reliée à l'entrée d'activation.

Qui plus est, on peut remarquer que l'entrée de retenue de l'additionneur doit être mise à 1 dans un seul cas: quand l'inverseur est actif. Vu que ces deux circuits doivent impérativement fonctionner ensemble, on peut fusionner les deux signaux censés les commander en un seul.



Signe-magnitude

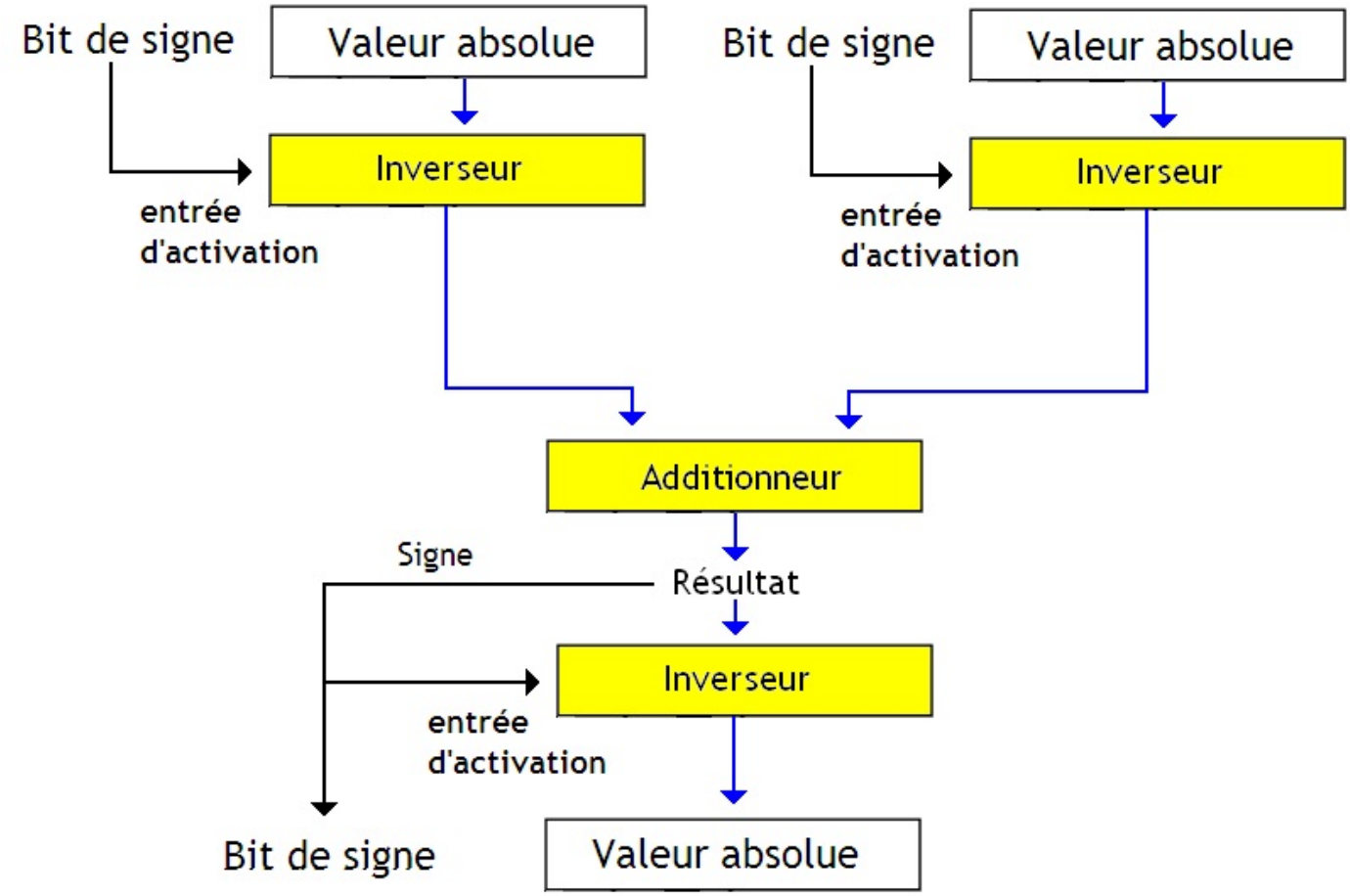
Maintenant que l'on sait effectuer des additions et des soustractions sur des nombres codés en complément à deux ou sur des nombres non-signés, on va voir ce qui se passe pour les nombres codés en signe--valeur absolue. La gestion de ces nombres est un peu plus compliquée à cause des bits de signe : on doit en tenir compte dans nos calculs. Par exemple, notre circuit doit

pouvoir additionner deux nombres positifs, mais aussi un négatif et un positif, ou deux négatifs.

Addition

Une solution simple consiste à convertir nos nombres codés en signe-valeur absolue vers du complément à un, faire l'addition en complément à un, et retraduire le tout en représentation signe-magnitude. Il nous faut donc un circuit capable de convertir les valeurs absolues de nos nombres en complément à un, et un autre pour traduire le résultat du calcul. Avec un additionneur en plus, bien sûr. Ces circuits de traduction sont de simples inverseurs commandables. Ces inverseurs sont identiques à celui vu au-dessus : ils disposent d'une entrée de commande qui dit d'inverser ou non leurs entrées. La commande des inverseurs devra être déduite des bits de signes. Sans compter qu'il faudra déduire le bit de signe du résultat.

Cela donne ce circuit :



En rusant un petit peu, on peut se passer d'un inverseur. Mais dans ce cas, le circuit devient plus compliqué.

Comparaison

Je tiens à signaler que les comparaisons sont souvent "fabriquées" à partir de soustractions. Pour comparer deux nombres, il suffit simplement de soustraire les deux nombres, de comparer le résultat avec zéro et de regarder le signe du résultat :

- si le résultat est positif, le nombre auquel on a soustrait l'autre est plus grand ;
- si le résultat est négatif, le nombre auquel on a soustrait l'autre est plus petit ;
- si le résultat est nul, les deux nombres sont égaux.

Une fois que l'on a fait ces tests, le résultat peut alors être oublié et n'a pas à être conservé. Il suffit juste de rajouter quelques circuits à base de portes ***XOR***, ***ET***, ***OU***, et ***NON*** à notre soustracteur, et relier ceux-ci au registre d'état pour le mettre à jour. Par exemple, pour tester si le résultat est nul, il suffit de regarder la valeur de ses bits : un résultat vaut zéro si et seulement si tous ses bits sont à zéro. Faire un vulgaire ***NOR*** sur tous les nombres du résultat permet ainsi de savoir si celui-ci est nul ou non. Pour voir si un résultat est positif ou négatif, il suffit de regarder son bit de signe (son bit de poids fort).

Multiplication

Après avoir vu quelques opérations simples, comme les décalages/rotations, les additions et les soustractions, il est temps de

passer à des opérations un peu plus gourmandes en terme de temps et de circuits. Nous allons aborder la multiplication, effectuée par un circuit nommé le **multiplieur**.

Entiers non-signés

Pour commencer, petite précision de vocabulaire : une multiplication s'effectue sur deux nombres, dont le premier est appelé **multiplicande**, et l'autre **multiplieur**. Dans ce qui va suivre, on va supposer que les deux sont positifs. Comme pour l'addition, nous allons calculer une multiplication de la même façon qu'on a appris à le faire en primaire, avec un petit détail : nous allons travailler en binaire. Pour effectuer une multiplication en binaire, on fait comme en décimal :

- on multiplie le multiplicande par le premier chiffre du multiplieur ;
- on recommence et on multiplie par le deuxième chiffre du multiplieur, mais en décalant le résultat d'un cran ;
- on recommence et on multiplie par le troisième chiffre, mais en décalant le résultat de deux crans ;
- on continue ainsi de suite jusqu'à avoir épuisé tous les chiffres du multiplieur... ;
- et enfin, on additionne tous les résultats temporaires obtenus lors des étapes du dessus.

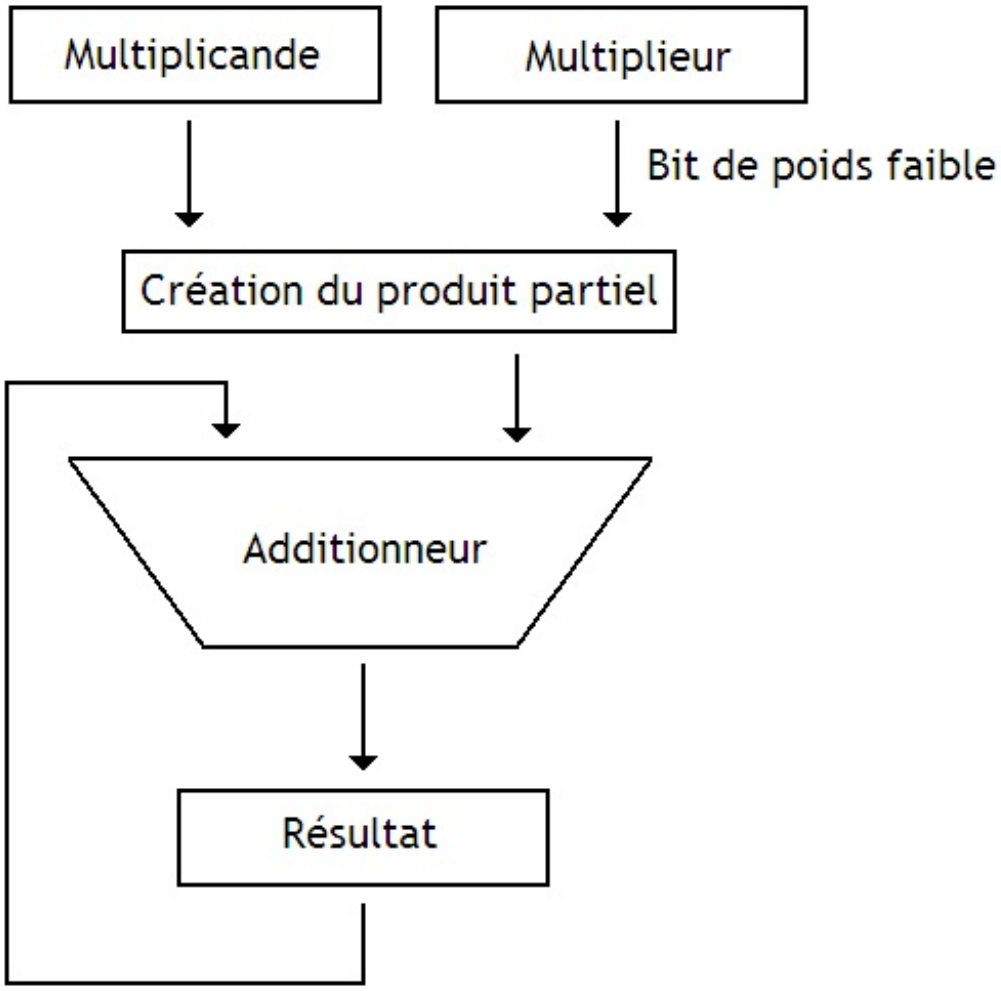
Exemple avec la multiplication de deux nombres de 4 bits A et B, composés respectivement des bits a3, a2, a1, a0 pour A et b3, b2, b1, b0 pour B.

$A \times B$				$a3$	$a2$	$a1$	$a0$
				$b3$	$b2$	$b1$	$b0$
Produit partiel n°1	0	0	0	$a3 \times b0$	$a2 \times b0$	$a1 \times b0$	$a0 \times b0$
Produit partiel n°2	0	0	$a3 \times b1$	$a2 \times b1$	$a1 \times b1$	$a0 \times b1$	0
Produit partiel n°3	0	$a3 \times b2$	$a2 \times b2$	$a1 \times b2$	$a0 \times b2$	0	0
Produit partiel n°4	$a3 \times b3$	$a2 \times b3$	$a1 \times b3$	$a0 \times b3$	0	0	0

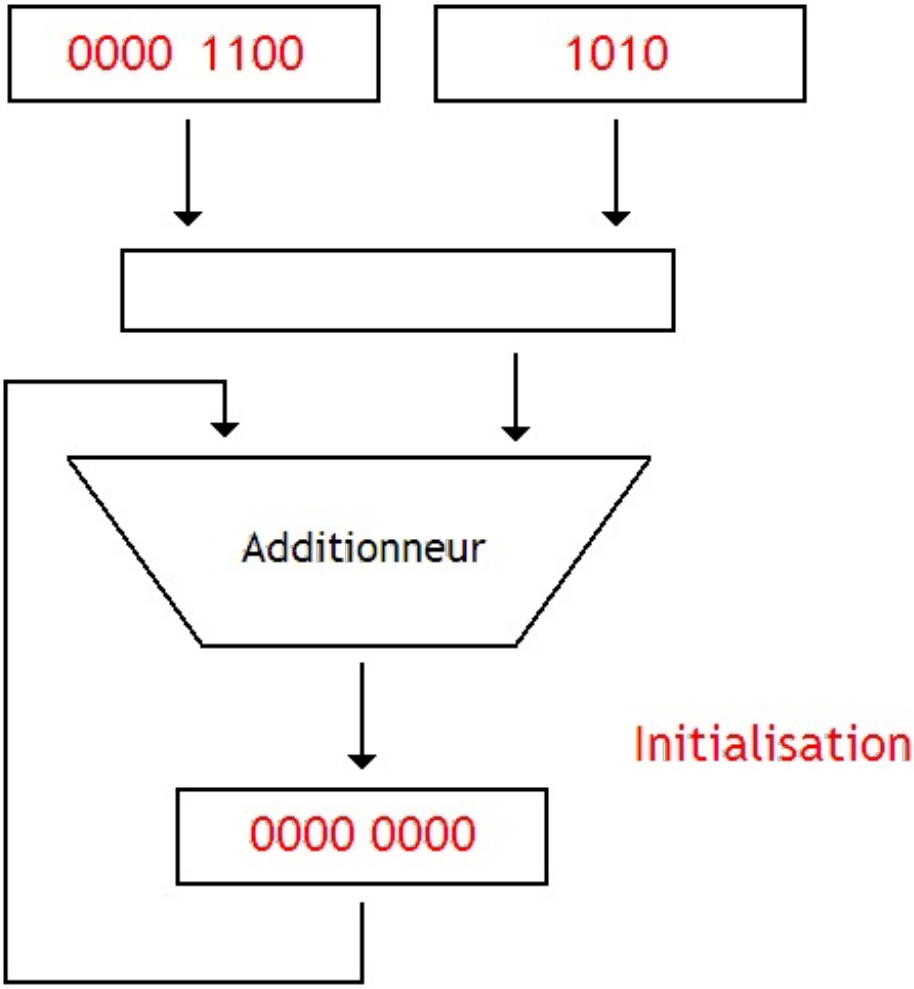
Comme on le voit, notre multiplication génère un grand nombre de résultats temporaires, chacun provenant de la multiplication de notre multiplicande par un chiffre du multiplieur, auquel on aura appliqué un décalage. Ces résultats temporaires sont appelés des **produits partiels**. Ce sont les lignes dans le tableau du dessus. Générer ces produits partiels nécessite donc de quoi multiplier des bits entre eux. Il suffira ensuite d'avoir un additionneur pour additionner tout ces produits partiels, et le tour est joué.

Circuit

Dans les multiplieurs les plus simples, on génère ces produits partiels les uns après les autres, et on les additionne au fur et à mesure qu'ils sont calculés. Pour cela, on utilise un registre qui stocke le résultat. Celui-ci est initialisé à zéro au commencement de la multiplication. De même, le multiplieur et le multiplicande sont aussi placés dans des registres. Dans ce qui va suivre, on effectuera notre multiplication de droite à gauche : on multiplie d'abord le multiplicande par le bit de poids faible du multiplieur, puis par le bit suivant, et ainsi de suite. Le circuit est le suivant :



Le fonctionnement de ce circuit est simple à comprendre. On commence par initialiser nos registres à leurs valeurs respectives. Ensuite, on génère le produit partiel et on l'additionne au registre résultat. Après cela, on décale le contenu du registre du multiplicande d'un cran vers la gauche, et on décale celui du multiplieur vers la droite. Et on recommence.



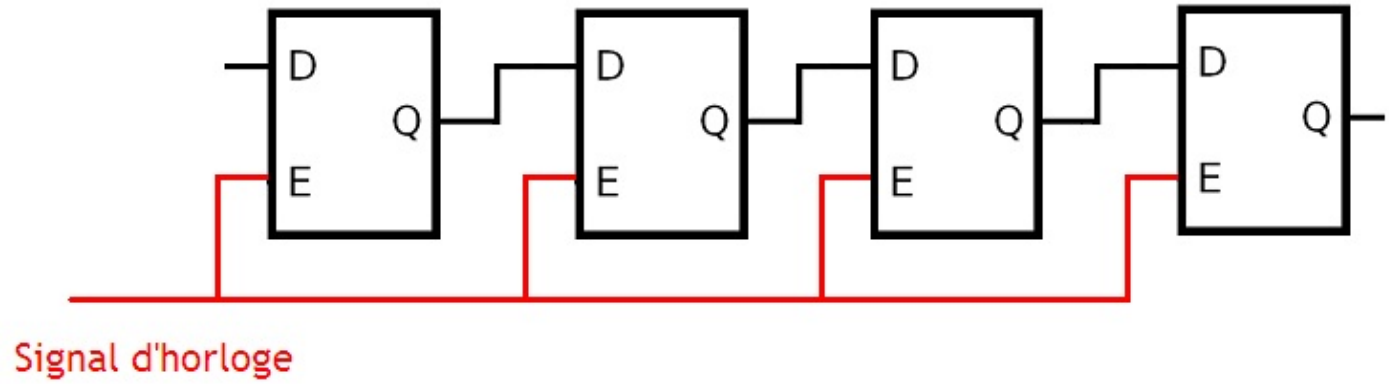
Bien sûr, cet enchainement d'additions doit se terminer quand tous les bits du multiplieur ont étés passés en revue. Pour terminer notre addition au bon moment, notre circuit doit contenir un petit compteur, qui contient le nombre de bits du multiplieur qu'il reste à traiter. Quand ce compteur atteint la bonne valeur, la multiplication est terminée. Il faut aussi prévoir un petit circuit qui se chargera de l'initialisation de nos registres.

Décalages automatiques

Comme vous l'avez remarqué, les deux registres du multiplicande et du multiplieur sont décalés d'un cran à chaque cycle d'horloge. Pour effectuer ce décalage automatique, on utilise ce qu'on appelle un registre à décalage. Un registre à décalage est un composant électronique qui fonctionne comme (j'ai pas dit qui est) un registre couplé à un décaleur, ce décaleur se chargeant de décaler les bits du nombre stocké dans le registre quand on lui demande.

L'implémentation la plus simple d'un registre à décalage consiste à prendre des bascules D, et à les relier en série : la sortie d'une bascule allant sur l'entrée de la suivante. Toutes ces bascules sont ensuite reliées à la même horloge. Ainsi, le contenu de ce registre est décalé d'un cran à chaque cycle d'horloge. On en déduit que ce circuit est tout de même lent : notre multiplication s'effectuera en autant de cycles qu'il y a de bits dans le multiplieur.

Exemple avec un registre qui décale d'un cran vers la droite



Génération des produits partiels

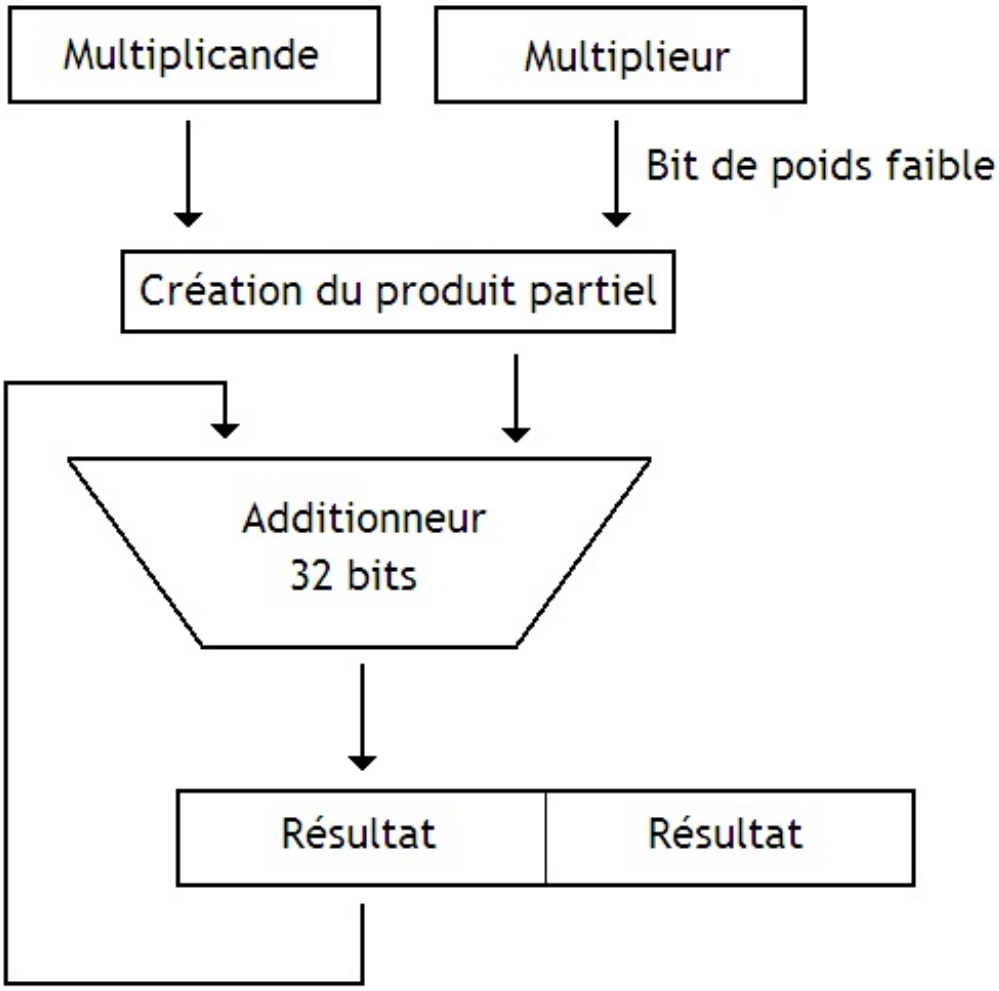
Générer notre produit partiel est très simple. Tout d'abord, on doit remarquer qu'une partie de cette génération s'est faite en décalant le contenu du registre du multiplicande. Il ne reste plus qu'à multiplier notre multiplicande par le bit de poids faible du multiplieur. Pour cela, on fait comme en décimal : on multiplie chaque bit du multiplicande par un bit du multiplieur. Pour cela, rien de plus simple vu que les tables de multiplication sont vraiment très simples en binaires : jugez plutôt !

Opération	Résultat
0×0	0
0×1	0
1×0	0
1×1	1

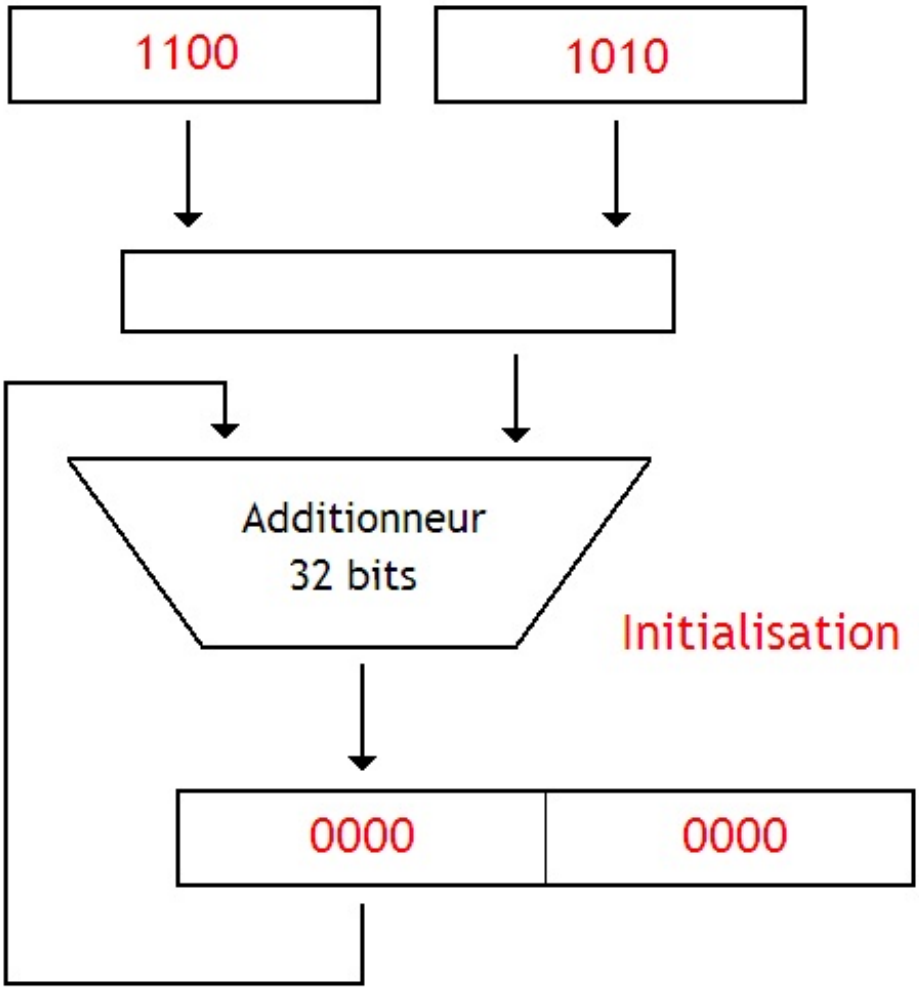
Quel dommage que l'on ne compte pas naturellement en binaire : vous n'auriez pas eus à vous farcir une dizaine de tables de multiplications complètement indigestes ! Cette table de vérité ressemble fortement à une table de vérité d'une porte ***ET***, et pour cause : c'est la table de vérité d'une porte ***ET*** ! Ainsi, notre circuit est donc très simple : il suffit d'effectuer un ET entre les bits du multiplicande, et le bit du multiplieur qu'on a sélectionné juste avant.

Inversion

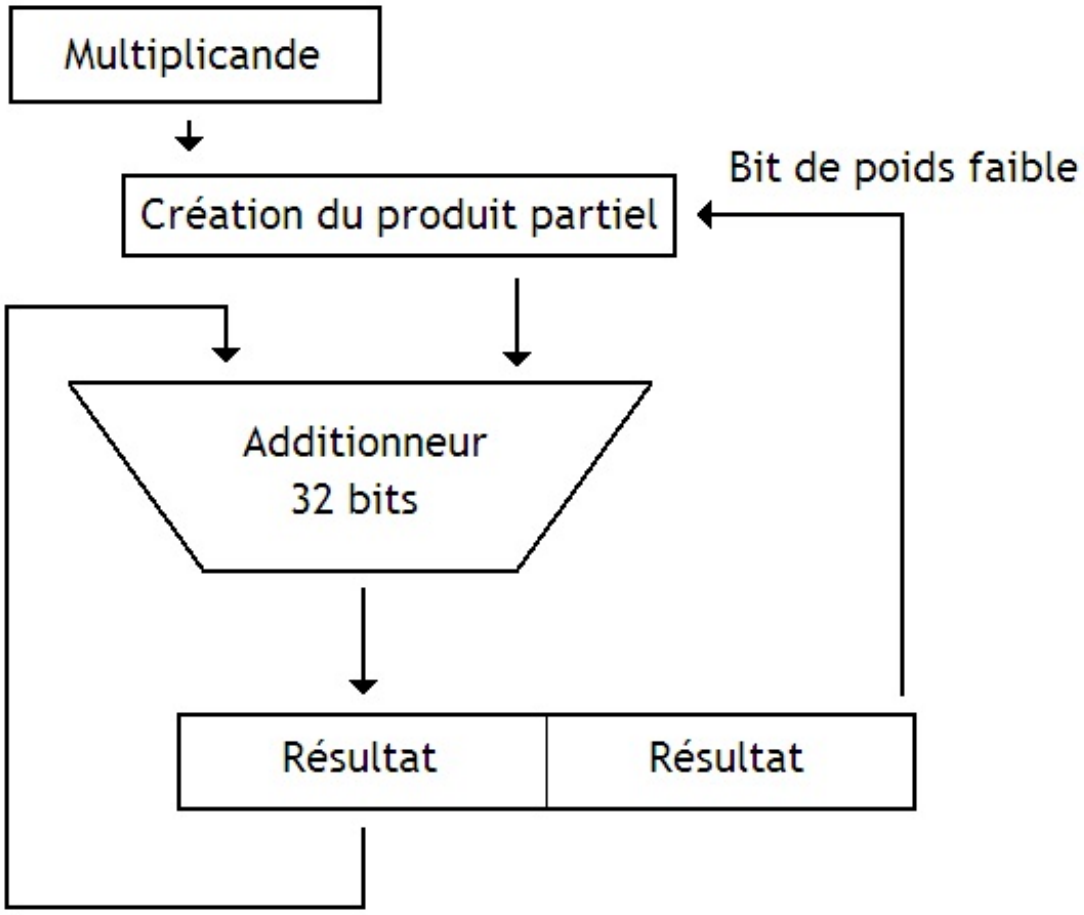
le circuit vu au-dessus est franchement améliorable. En réfléchissant bien, on peut trouver une petite astuce assez sympathique qui permet de gagner pas mal en circuits. Avec le circuit du haut, on stockait le résultat de l'addition dans les bits de poids faible du registre du résultat. Sachez qu'on peut aussi le stocker dans les bits de poids forts, et décaler ce résultat d'un cran à droite à chaque cycle. Cela donnera le même résultat. On se retrouve alors avec un circuit un peu différent : cette fois, le multiplicande n'est pas décalé à chaque cycle. Mais le résultat le sera à sa place, ainsi que le multiplieur (cela ne change pas) : les deux étant décalés vers la droite.



Prenons un exemple : on veut multiplier deux nombres de 32 bits. Avec la technique du dessus, on devrait utiliser des additionneur 64 bits, et un registre résultat de 64 bits. Mais avec ce nouveau circuit, on peut se contenter d'un additionneur 32 bits. On gagne ainsi pas mal en circuits.



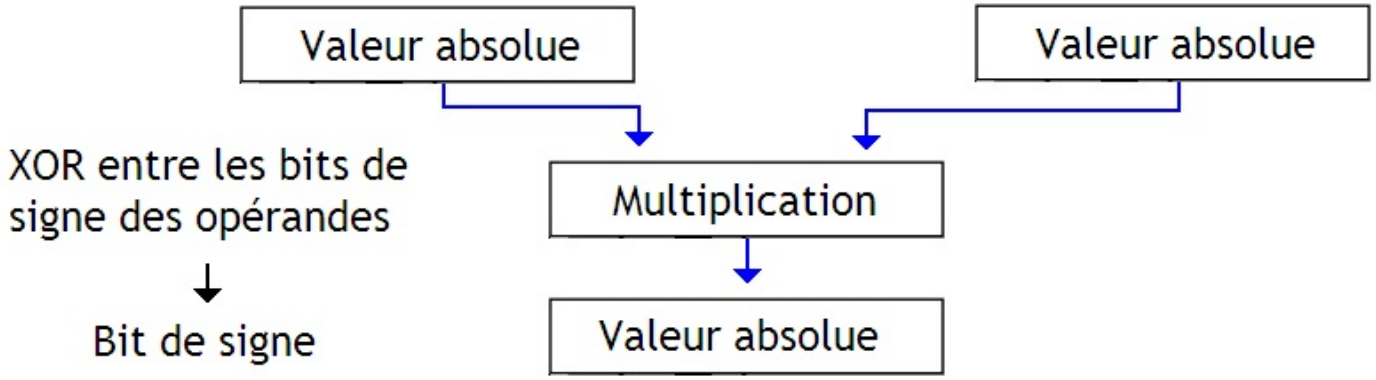
Il est même possible de ruser encore plus : on peut se passer du registre pour le multiplieur. Il suffit pour cela d'initialiser les bits de poids faible du registre résultat avec le multiplieur au démarrage de la multiplication, et de prendre le bit de poids faible du résultat.



Entiers signés

Tous les circuits qu'on a vu plus haut sont capables de multiplier des nombres entiers positifs. Mais nous n'avons pas encore vu comment traiter des entiers signés.

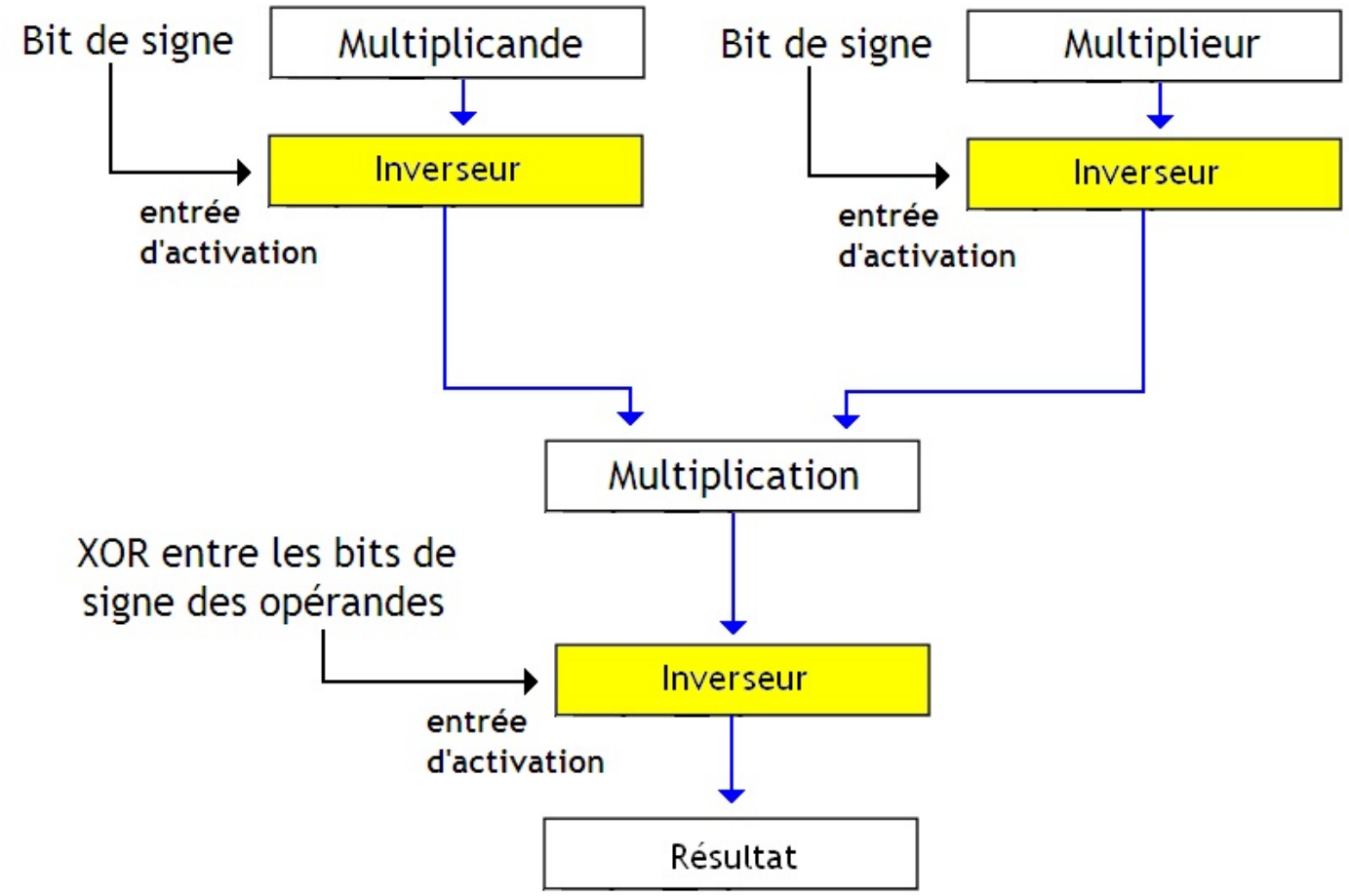
Commençons par le cas le plus simple : les entiers en signa-valeur absolue. Pour eux, la multiplication est très simple : il suffit de multiplier les valeurs absolues, et de déduire le bon signe. La multiplication des valeurs absolues peut s'effectuer avec les circuits vus au-dessus sans aucun problème. Quand à la détermination du signe, il s'agit d'un vulgaire XOR entre les bits de signe des deux nombres à multiplier.



Pour les nombres codés en complément à 1, la solution se base sur le même principe. Comme avec les nombres en signe-valeur absolue, on va multiplier les valeurs absolues des deux opérandes, et on va en déduire le signe en fonction des signes des deux opérandes. On va donc devoir calculer la valeur absolue du multiplicande et de multiplieur, les multiplier, et éventuellement inverser le résultat si besoin.

le calcul des valeurs absolues des opérandes s'effectue avec un inverseur commandable. Si l'opérande est négative, on actionne cet inverseur pour qu'il inverse notre opérande : on obtient bien la valeur absolue. Ensuite, on effectue la multiplication, et on

traite le résultat pour qu'il aie le bon signe. Cela se fait en inversant le résultat s'il doit être négatif, et en faisant rien sinon.



Pour la multiplication en complément à deux, les choses se compliquent. Tenter de multiplier les valeurs absolues et de corriger le résultat est une solution, mais obtenir la valeur absolue d'un nombre en complément à deux nécessitera l'intervention d'un additionneur. Le circuit qu'on obtiendrait serait alors un peu trop complexe. Nous allons donc devoir adapter notre circuit pour qu'il gère les multiplieurs et multiplicandes négatifs. Pour commencer, nous pouvons vous faire une petite remarque : les circuits vus au-dessus fonctionnent parfaitement quand les deux opérandes sont négatives. Elles donnent alors le bon résultat, et il n'y a rien à faire. Reste à gérer les autres situations.

Multiplicande négatif

Nous allons commencer par regarder ce qui se passe quand le multiplicande est négatif, et le multiplieur positif.

$A \times B$					1	1	0	1
					0	1	0	1
Produit partiel n°1					1	1	0	1
Produit partiel n°2				0	0	0	0	
Produit partiel n°3			1	1	0	1		
Produit partiel n°4		0	0	0	0			

Maintenant, regardez les produits partiels 1 et 3. Ce sont des " copies " du multiplicande, codées sur 4 bits, qu'on a décalées d'un ou plusieurs crans vers la gauche. Seul problème : ce multiplicande est censé être un entier négatif. Hors, on se retrouve avec un vides à gauche de ces produits partiels : le produit partiel est codé sur moins de bits que le résultat. Avec l'algorithme d'avant, ces vides étaient remplis avec des zéros : les produits partiels étaient devenus positifs, au lieu d'être négatifs ! D'où un résultat

faux.

Pour résoudre ce problème, il suffit de remplir les vides à gauche du produit partiel par la bonne valeur, afin de traduire notre produit partiel en un entier suffisamment long pour remplir totalement un produit partiel. Cette conversion d'un entier codé en complément à deux en un autre entier, codé sur plus de bits s'appelle la *Sign Extension*. Si vous vous rappelez le premier chapitre, vous vous souvenez que pour effectuer cette conversion, on doit remplir ces vides par le bit de signe du nombre pour obtenir un résultat correct.

$A \times B$					1	1	0	1
					0	1	0	1
Produit partiel n°1	1	1	1	1	1	1	0	1
Produit partiel n°2	0	0	0	0	0	0	0	
Produit partiel n°3	1	1	1	1	0	1		
Produit partiel n°4	0	0	0	0	0			

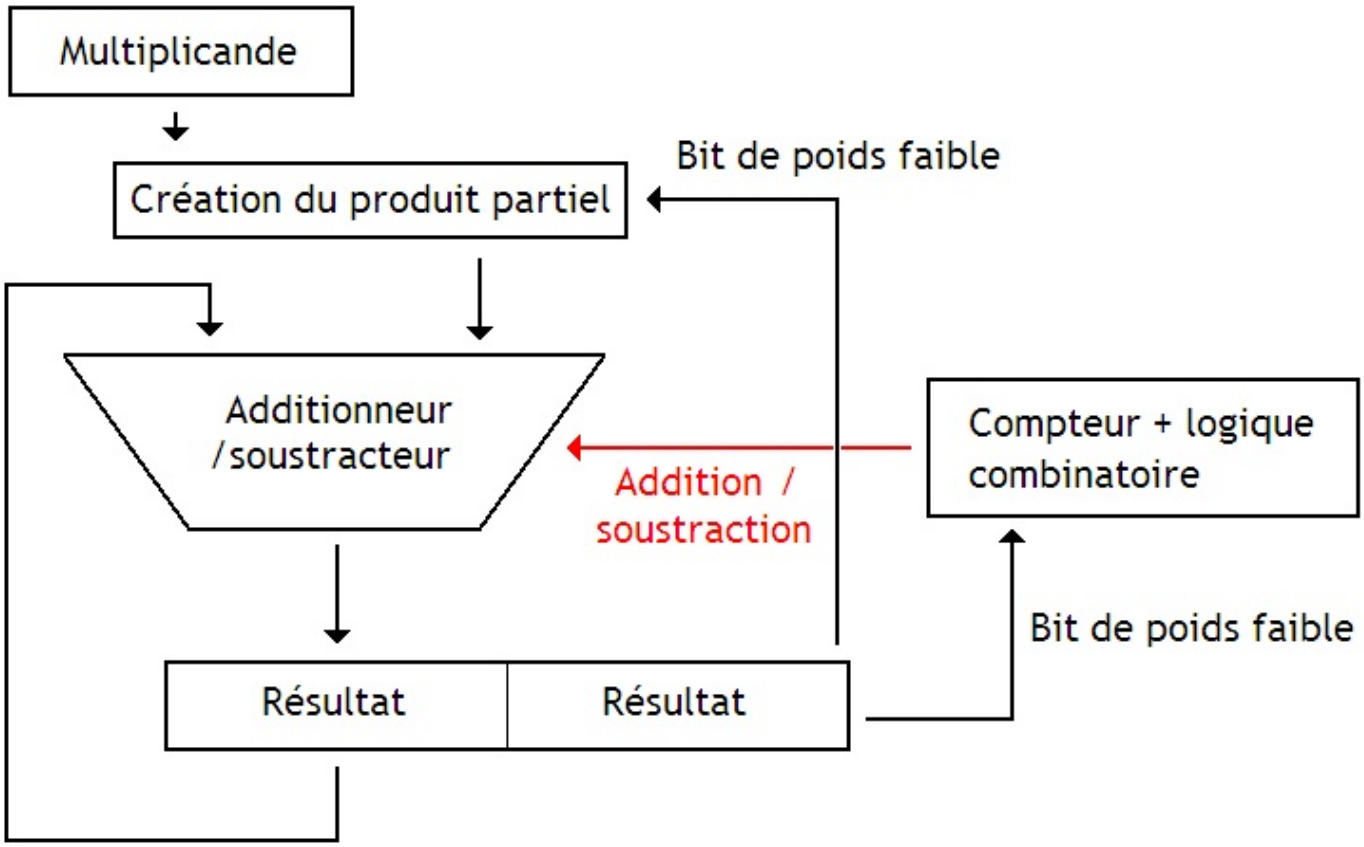
Idéalement, cette conversion doit se faire entre la génération du produit partiel, et l'addition. Avec le premier circuit, qui décale automatiquement le multiplicande, cette *Sign Extension* n'est pas faite automatiquement, et doit être effectuée par un circuit supplémentaire. Mais dans ce cas, la *Sign Extension* est très dure à effectuer : il faut se souvenir où est le bit de signe, vu que le multiplicande est décalé automatiquement à chaque cycle d'horloge. Et cela utilise un compteur.

Par contre, les circuits qui ne modifient pas les multiplicande permettent d'effectuer celle-ci beaucoup plus simplement. Il suffit simplement de faire en sorte que le décalage du résultat soit un décalage arithmétique. On doit donc modifier quelque peu le registre à décalage qui stocke le résultat, et tout fonctionnera à merveille.

Multiplieur négatif

Pour traiter le cas d'un multiplieur négatif, le circuit vu au-dessus ne fonctionne pas parfaitement. Mais il y a moyen de le corriger pour qu'il fonctionne à merveille. L'idée est simple : si le multiplieur est négatif, on ne va pas ajouter le produit partiel calculé à partir du bit de signe du multiplieur. A la place, on va le soustraire. C'est magique : on a juste à faire cela, et ça marche ! Et oui, que vous le vouliez ou non, c'est comme cela. On peut se demander pourquoi cela fonctionne, mais l'explication est assez mathématique et franchement dure à comprendre, aussi je vous épargne les détails.

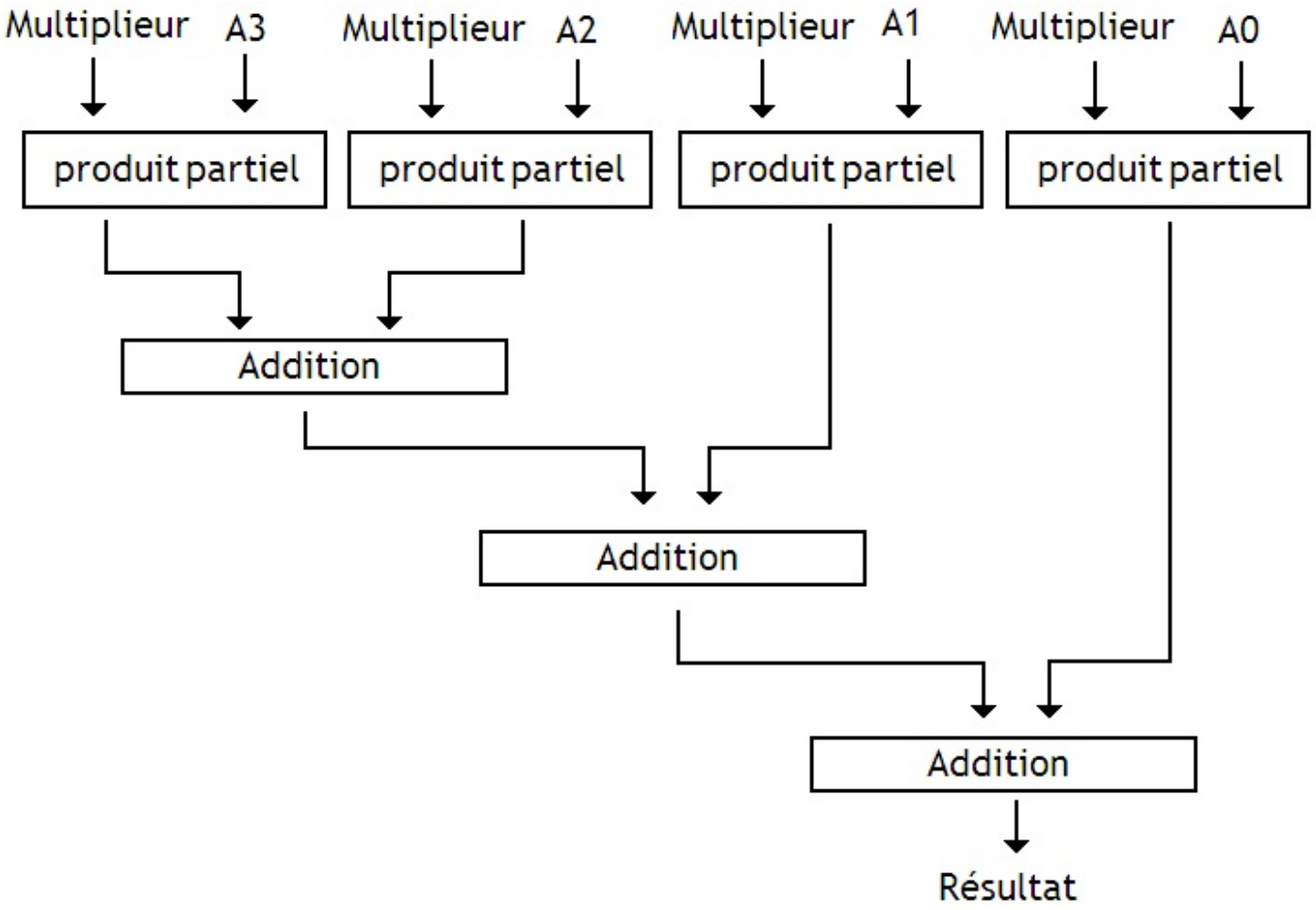
Pour adapter notre circuit, il suffit d'ajouter un circuit combinatoire au compteur intégré au circuit : ce circuit servira à détecter le produit partiel à inverser, et demandera à notre additionneur soustracteur d'effectuer ou non une soustraction.



Array Multipliers

Disons les choses franchement : les circuits vus au-dessus sont des bouses totalement innommables. La raison : ils sont lents ! La raison à cela est très simple : ces multiplieurs calculent et additionnent les produits partiels uns par uns, au rythme d'un produit partiel par cycle d'horloge. Il y a tout de même moyen de faire mieux. Au lieu de calculer tous les produits partiels uns par uns et les additionner au même rythme, on peut les calculer en parallèle. Pour cela, rien de plus simple : on a juste à créer un circuit qui calcule ces produits partiels en parallèle, et qui les additionne.

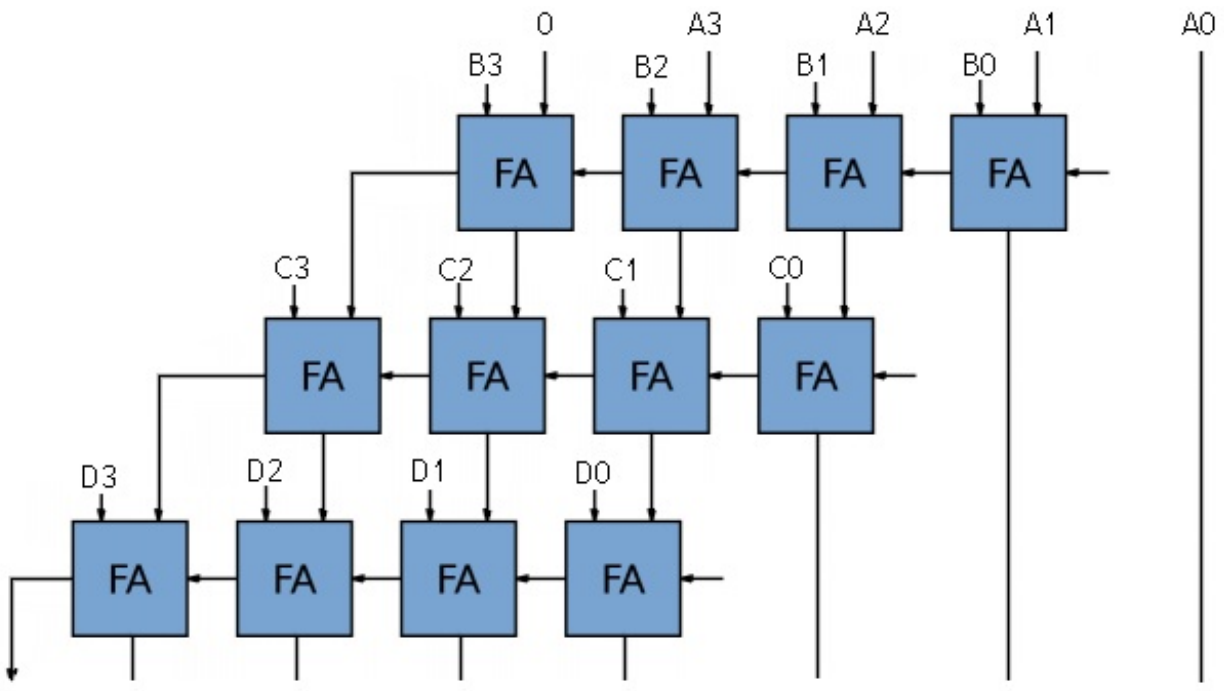
Dans sa version la plus simple, notre circuit va simplement enchainer ses additionneurs les uns à la suite des autres.



Pour implémenter cette technique, on peut calculer le produit partiel de deux façons. On peut utiliser des portes ET, dont les sorties sont relié à un décaleur par 1, 2, 4, etc suivant le produit partiel. Mais on peut aussi se passer du décaleur en rusant un peu sur le câblage du circuit et en mettant certaines entrées des additionneurs à zéro.

Ripple Carry Adder Tree

Dans sa version la plus simple, on peut utiliser des additionneurs à propagation de retenue pour créer notre multiplieur. Pour montrer à quoi pourrait ressembler un tel circuit, on va prendre l'exemple de la multiplication de deux nombres de 4 bits. On aura alors 4 produits partiels à additionner, ce qui se fera avec le circuit suivant :



On pourrait penser qu'utiliser des additionneurs aussi lents serait un désavantage. Mais curieusement, cela ne nuit pas trop aux performances du multiplieur. Utiliser des additionneurs à anticipation de retenue ou des additionneurs à sélection de retenue donnerait des gains relativement faibles. Par contre, utiliser des additionneurs à propagation de retenue permet d'économiser beaucoup de portes logiques et de transistors.

Carry Save Array

Il est évident qu'utiliser des additionneurs à propagation de retenue n'est pas optimal. Mais comment faire mieux ? Il faut bien se souvenir que le problème auquel on fait face est d'additionner plusieurs nombres à la suite. Pour résoudre ce problème, les chercheurs ont cherché une solution et on fini par inventer une nouvelle représentation des nombres, particulièrement adaptée aux additions successives. Cette représentation est ce qu'on appelle la représentation ***Carry Save***.

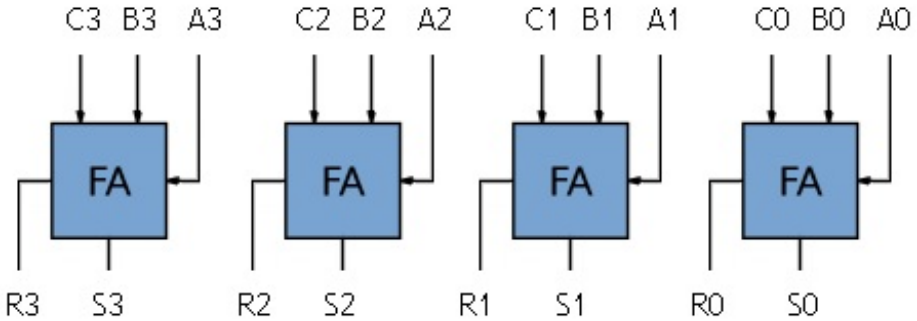
Avec cette représentation, on va pouvoir additionner plusieurs nombres et obtenir un résultat représenté en *Carry Save*. Une fois ce résultat obtenu, on peut le convertir en binaire normal avec un additionneur conventionnel. Pour effectuer cette addition en *Carry Save*, on additionne les bits de même poids, et on stocke le résultat sur plusieurs bits, sans propager les retenues.

Par exemple, regardons ce qui se passe avec trois nombres. L'addition de trois bits en *Carry Save* va donner un résultat sur deux bits : une retenue, et une somme. En additionnant chaque bit de nos trois nombres uns par uns, on obtient la somme des trois nombres en *Carry Save*.

$$1\ 0\ 0\ 0 + 1\ 0\ 1\ 0 + 1\ 1\ 1\ 0 = 3\ 1\ 2\ 0 = 11\ 01\ 10\ 00$$

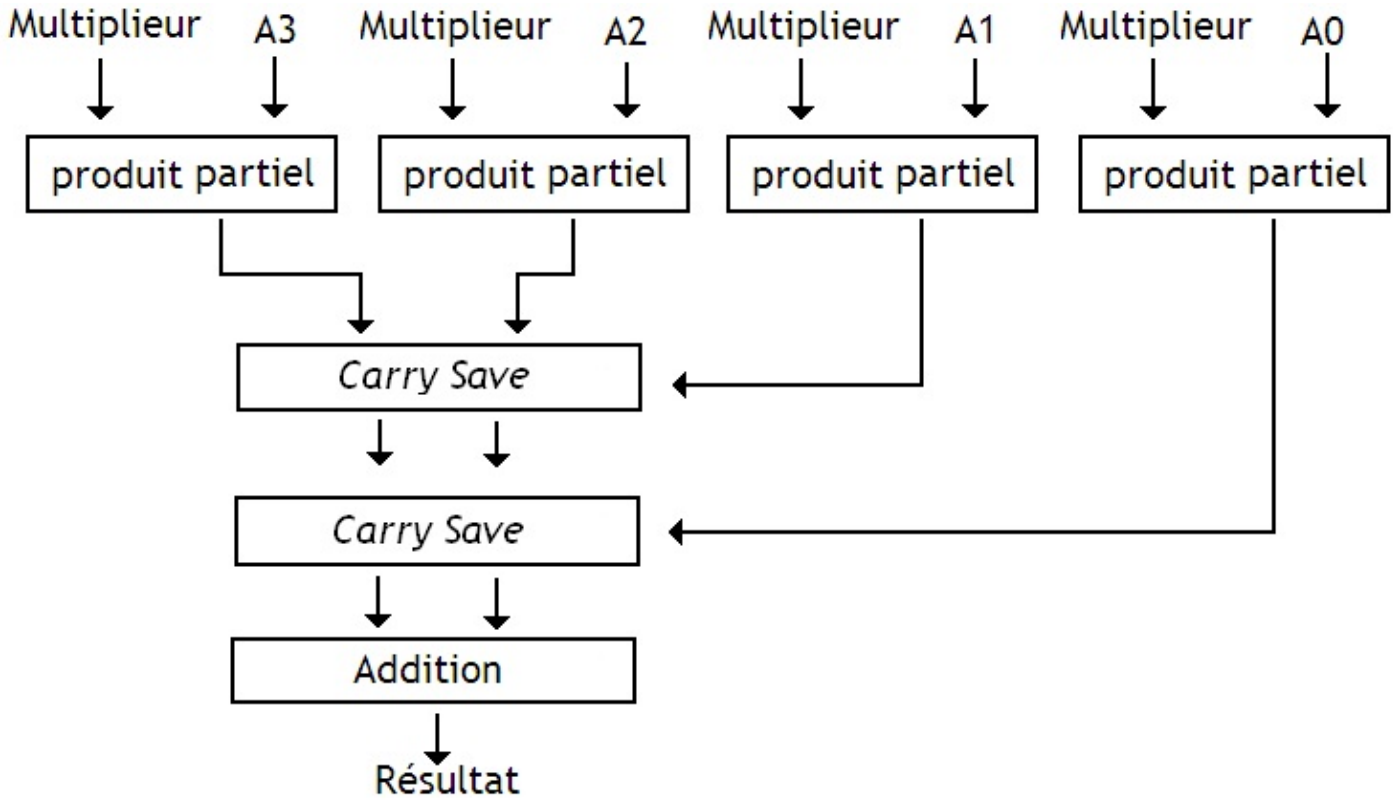
Le résultat peut être interprété comme étant composé de deux résultats : un nombre composé des sommes des bits, et un autre composé des retenues. En additionnant convenablement les deux, on peut retrouver le résultat de l'addition, codé en binaire normal.

Cette addition de trois nombres en *Carry Save* n'est pas compliquée à faire. Il suffit de créer un petit circuit capable d'additionner trois bits et d'en placer plusieurs les uns à coté des autres. Or, on connais déjà ce circuit capable d'additionner trois bits : c'est l'additionneur complet. On obtient alors le circuit ci-dessous :

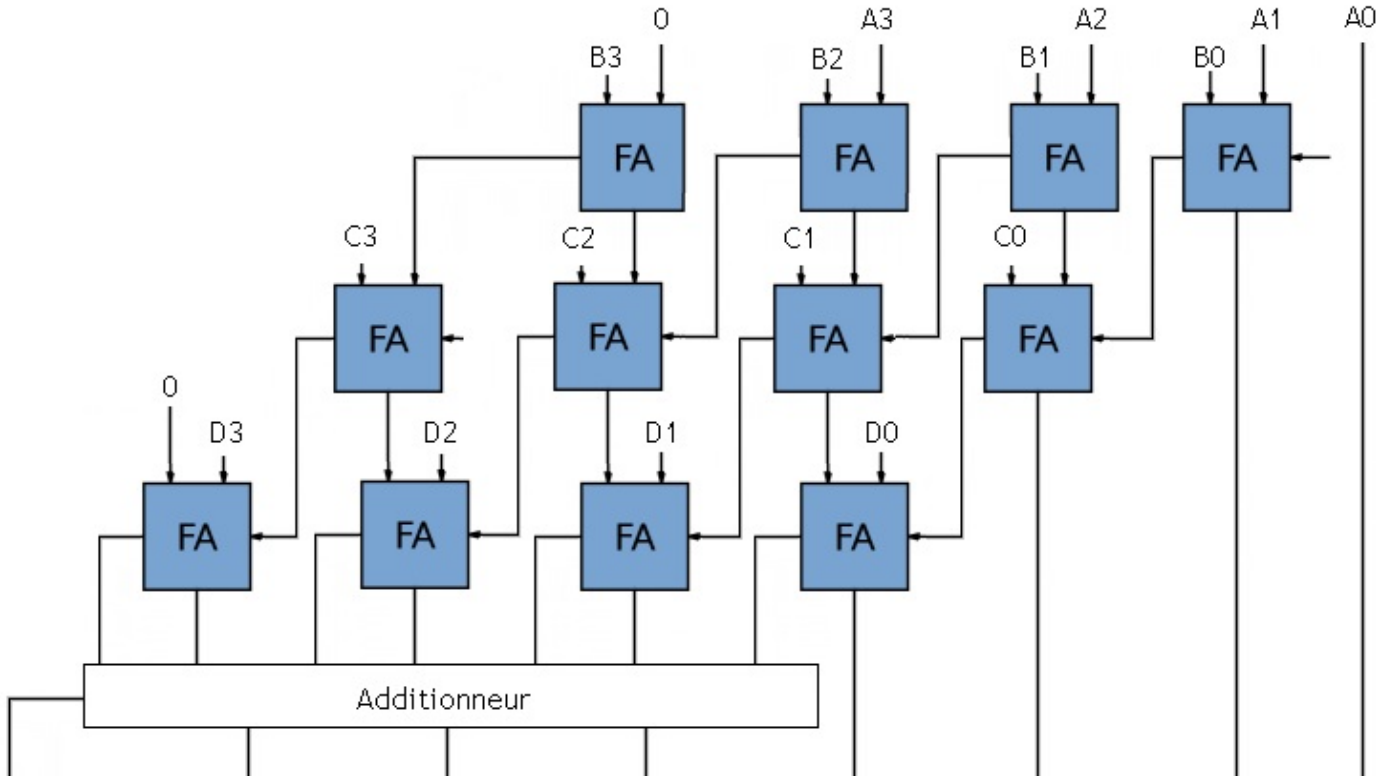


Ainsi, au lieu d'utiliser deux additionneurs normaux, on utilise un additionner *Carry Save*, et un additionneur normal. L'additionneur *Carry Save* étant bien plus rapide que tout autre additionneur, on gagne beaucoup en performances.

Mais le même principe peut être adapté pour la somme de trois, quatre, cinq nombres ou plus. Cela peut se faire de diverses facons, mais la plus simple consiste à réutiliser notre additionneur *Carry Save* à trois opérandes. Il suffit d'en enchainer plusieurs les uns après les autres, pour additionner nos produits partiels.

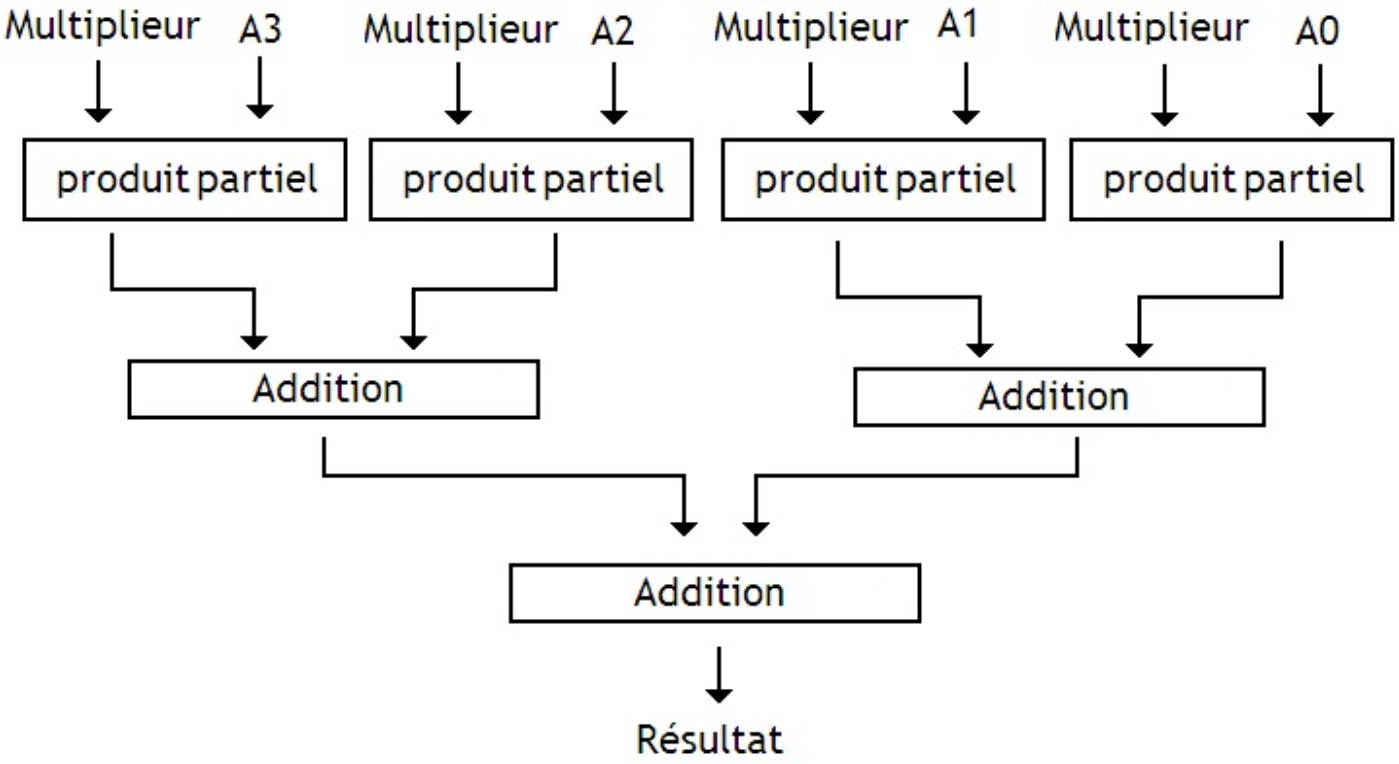


On obtient alors ce circuit :



Tree Multipliers

Les additionneurs vus au-dessus peuvent encore subir quelques améliroations. Tou d'abord, il faut savoir qu'enchainer les additionneurs les uns à la suite des autres n'est pas la meilleur solution. Le mieux est de les organiser comme ceci :



Avec cette organisation "en arbre", on arrive à effectuer certaines additions en parallèles d'autres, ce qui permet de gagner du temps. Il existe divers types d'organisations en arbres, dont les deux plus connues sont les arbres de Wallace, des arbres Dadda. Ces arbres utilisent tous des additionneurs *Carry-Save*.

Division

Après la multiplication, nous allons voir comment effectuer des divisions. Autant prévenir tout de suite : la division est une opérations très complexe et particulièrement lente, bien plus qu'une addition ou une multiplication. Pour information, sur les processeurs actuels, la division est entre 20 à 80 fois plus lente qu'une addition/soustraction, et presque 7 à 26 fois plus lente qu'une multiplication.

Mais on a de la chance : c'est aussi une opération assez rare. Un programme effectue rarement des divisions, les plus rares étant les divisions entières tandis que les divisions les plus fréquentes sont les divisons entre deux nombres flottants.

Souvent, les divisions les plus couramment utilisées dans un programme sont des divisions par une constante : un programme devant manipuler des nombres décimaux aura tendance à effecteur des divisons par 10, un programme manipulant des durées pourra faire des divisions par 60 (gestion des minutes/secondes) ou 24 (gestion des heures). Diverses astuces permettent de remplacer ces opérations de divisions par des suites d'instructions plus simples mais donnant le même résultat.

J'ai parlé plus haut des décalages, qui permettent de remplacer de divisons par 2^n. Mais il existe d'autres méthodes, qui fonctionnent pur un grand nombre de constantes. Par exemple, on peut remplacer une division par une constante par une multiplication un peu bizarre : la multiplication par un entier réciproque).

Sachant cela, certains processeurs ne possèdent pas d'instruction de division. Inclure une instruction de division n'accélérerait qu'un faible nombre d'instructions, et ne donnerait pas lieu à des gains assez importants en terme de performance : accélérer 1% des instructions d'un programme (ici, les divisions) en implémentant un circuit complexe et gourmand en transistors alors qu'on pourrait utiliser ces circuits pour câbler des instructions plus utiles serait du gâchis. Certains processeurs implémentent toutefois la division dans une instruction machine, disposant souvent d'un circuit dédié. Les gains ne sont pas forcément faramineux, mais ne sont pas forcément négligeables non plus.

Division à restauration

L'algorithme le plus simple que l'on puisse créer pour exécuter une division consiste à faire la division exactement comme en décimal, mais d'une faon un peu différente.

```
 100011001111 | 111
- 111         |-----------
-------------- | 101000010
    111       |
  - 111       |
  -----------  |
        111    |
      - 111    |
      -------  |
         01    |
```

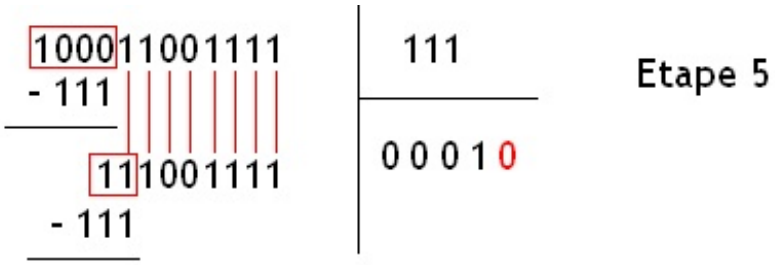
Algorithme

Prenons un exemple. Nous allons cherche à diviser 100011001111 (2255 en décimal) par 111 (7 en décimal). Pour commencer, nous allons commencer par sélectionner le bit de poids fort du dividende (le nombre qu'on veut diviser par le diviseur), et voir combien de fois on trouve le diviseur dans ce bit. Pour ce faire, on soustraire le diviseur à ce bit, et voir le signe du résultat. Si le résultat de cette soustraction est négatif, alors le diviseur est plus grand que ce qu'on a sélectionné dans notre dividende. On place alors un zéro dans le quotient. Dans notre exemple, cela fait zéro : on pose donc un zéro dans le quotient.

```
     100011001111 | 111
 - 111            |-----------
 -----            | 0
 Négatif          |
```

Ensuite, on abaisse le bit juste à coté du bit qu'on vient de tester, et on recommence. On continue ainsi tant que le résultat de la soustraction obtenue est négatif.

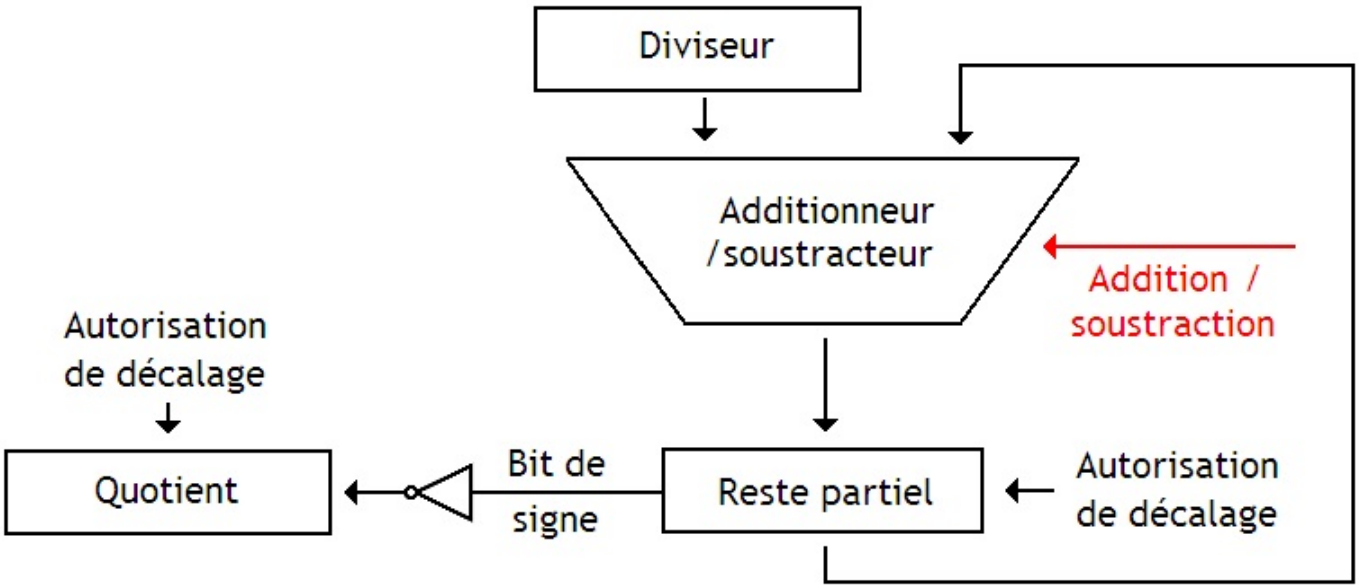
Quand le résultat de la soustraction n'est pas négatif, on met un 1 à la droite du quotient, et on recommence en partant du reste. Et on continue ainsi de suite.



Cette méthode s'appelle la **division avec restauration**.

Circuit

Notre algorithme semble se dessiner peu à peu : on voir qu'on devra utiliser des décalages et des soustractions, ainsi que des comparaisons. L'implémentation de cet algorithme dans un circuit est super simple : il suffit de prendre trois registres : un pour conserver le "reste partiel" (ce qui reste une fois qu'on a soustrait le diviseur dans chaque étape), un pour le quotient, et un pour le diviseur. L'ensemble est secondé par un additionneur/soustracteur, et par un peu de logique combinatoire. Voici ce que cela donne sur un schéma (la logique combinatoire est omise).



Notre algorithme se déroule assez simplement. Tout d'abord, on initialise les registres, avec le registre du reste partiel qui est initialisé avec le dividende.

Ensuite, on soustrait le diviseur de ce "reste" et on stocke le résultat dans le registre qui stocke le reste. Deux cas de figure se présentent alors : le reste partiel est négatif ou positif. Dans les deux cas, on réussit trouver le signe du reste partiel en regardant simplement le bit de signe du résultat. Reste à savoir quoi faire.

- Le résultat est négatif.
- En clair, cela signifie que le reste est plus petit que le diviseur et qu'on aurait pas du soustraire. Vu que notre soustraction a été effectuée par erreur, on doit remettre le reste tel qu'il était. Ce qui est fait en effectuant une addition. Il faut aussi mettre le bit de poids faible du quotient à zéro et le décaler d'un rang vers la gauche.
- Le résultat est positif.
- Dans ce cas, on met le bit de poids faible du quotient à 1, puis on décale celui-ci et on ne fait rien de plus.

Ensuite, il faut encore décaler le reste partiel. On décale de reste partiel pour mettre le diviseur à la bonne place sous le reste partiel lors des soustractions. Et on continue ainsi de suite jusqu'à ce que le reste partiel soit inférieur au diviseur.

Division sans restauration

La méthode précédente a toutefois un léger défaut : on a besoin de remettre le reste comme il faut lorsqu'on a soustrait le diviseur du reste alors qu'on aurait pas du et que le résultat obtenu est négatif. On fait cela en rajoutant le diviseur au reste. Et il y a moyen de se passer de cette restauration du reste partiel à son état originel.

On peut très bien continuer de calculer avec ce reste faux, pour ensuite modifier le quotient final obtenu de façon simple, pour obtenir le bon résultat. Il suffit simplement de multiplier le quotient par deux, et d'ajouter 1. Ça parait vraiment bizarre, mais c'est ainsi. Cette méthode consistant à ne pas restaurer le reste comme il faut et simplement bidouiller le quotient s'appelle la **division sans restauration**.

La division SRT

On peut encore améliorer cette méthode en ne traitant pas notre dividende bit par bit, mais en le manipulant par groupe de deux, trois, quatre bits, voire plus encore. Ce principe est (en partie) à la base de l'algorithme de **division SRT**. C'est cette méthode qui est utilisée dans les circuits de notre processeur pour la division entière.

Sur certains processeurs, le résultat de la division de deux groupes de bits est pré-calculé et stocké dans une petite mémoire : pas besoin de le recalculer à chaque fois avec un circuits, il suffit juste de lire cette mémoire, ce qui va beaucoup plus vite ! Pour information, on peut signaler que sur les processeurs les plus récents à l'heure où j'écris ce tutoriel, on peut traiter au maximum 4 bits à la fois. C'est notamment le cas sur les processeurs Core 2 duo.

Bien sûr, il faut faire attention quand on remplit cette mémoire : si vous oubliez certaines possibilités ou que vous y mettez des résultats erronés, vous obtiendrez un quotient faux pour votre division. Et si vous croyez que les constructeurs de processeurs n'ont jamais fait cette erreur, vous vous trompez : cela arrive même aux meilleurs !

Intel en a d'ailleurs fait les frais sur le Pentium 1. L'unité en charge des divisions flottantes utilisait un algorithme similaire à celui vu au-dessus (les mantisses des nombres flottants étaient divisées ainsi), et la mémoire qui permettait de calculer les bits du quotient contenait quelques valeurs fausses. Résultat : certaines divisions donnaient des résultats incorrects !
Fabriquer ces circuits de calcul n'est pas une mince affaire et les constructeurs de processeurs, ainsi que des chercheurs en arithmétique des ordinateurs, travaillent d'arrache-pied pour trouver des moyens de rendre nos circuits plus rapides et plus économes en énergie. Autant vous dire que les circuits que vous venez de voir sont vraiment des gamineries sans grande importance comparé à ce que l'on peut trouver dans un vrai processeur commercial !

Partie 4 : Mémoires

Comme son nom l'indique, ce chapitre va tenter de vous expliquer ce qu'est une mémoire et comment elle fonctionne. Nous allons voir en détail ce qu'est une mémoire, comment fait-elle pour retenir des bits, et quelles sont les améliorations apportées sur les mémoires récentes.

Mémoires

Mémoire.

Ce mot signifie dans le langage courant le fait de se rappeler quelque chose, de pouvoir s'en souvenir. La mémoire d'un ordinateur fait exactement la même chose (vous croyez qu'on lui a donné le nom de mémoire par hasard ?) mais dans notre ordinateur. Son rôle est donc de retenir que des données stockées sous la forme de suites de bits, afin qu'on puisse les récupérer si nécessaire et les traiter.

Des mémoires en veux-tu, en voilà !

Maintenant qu'on a la définition d'une mémoire, autant prévenir tout ce suite : toutes les mémoires ne sont pas faites de la même façon et il en existe différents types, chacun ayant ses avantages et ses inconvénients. Dans cette partie, on va passer en revue les différences les plus importantes.

Capacité mémoire

Pour commencer, on va commencer par enfoncer des portes ouvertes : on ne peut pas stocker autant de données qu'on veut dans une mémoire. Vous avez surement déjà du en faire l'expérience : qui n'a jamais eu un disque dur, une disquette, un CD-ROM ou DVD, ou une clé USB pleine ? Et ce qui vaut pour les mémoires que je viens de citer au-dessus marche pour toutes les mémoires.

Et à ce petit jeu là, toutes les mémoires ne sont pas égales : certaines peuvent contenir plus de données que d'autres. C'est la première différence entre nos mémoires : **la quantité de données qu'elles peuvent stocker**. Vu que toutes nos données sont stockées sous la forme de suites de bits, on peut facilement évaluer la capacité d'une mémoire à stocker un nombre plus ou moins de données : il suffit de compter le nombre maximal de bits qu'elle peut contenir. Ce nombre de bits que notre mémoire peut stocker porte un nom : c'est la **capacité** de la mémoire. Pour résumer, nos mémoires n'ont pas la même capacité, qui est le nombre maximal de bits qu'elle peut stocker.

Dans la majorité des mémoires, les bits sont regroupés en "paquets" contenant une quantité fixe de bits : des **"cases mémoires"**, aussi appelées ***bytes***.

Généralement, nos mémoires utilisent un *byte* de 8 bits. Autrefois, certaines mémoires avaient des cases mémoires de 6 ou 5 bits, parfois plus. Mais maintenant, la situation s'est un peu normalisée et la grosse majorité des mémoires utilisent un byte de 8 bits. Au fait : un groupe de 8 bits s'appelle un **octet**.

Hé, une minute ! Je croyais que *byte* et octet c'était la même chose ?

Rassurez-vous cher lecteur, c'est une erreur courante. Il faut vraiment connaitre le truc pour ne pas se faire avoir. En effet, *un byte n'est pas un octet* ! Vu que de nos jours nos mémoires utilisent des *bytes* d'un octet, on utilise souvent les deux termes de façon interchangeable. Mais ça reste tout de même un **abus de langage**.

Le fait que nos mémoires aient presque toutes des bytes faisant un octet nous arrange pour compter la capacité d'une mémoire. Au lieu de compter cette capacité en bits, on préfère mesurer la capacité d'une mémoire en donnant le nombre d'octets que celle-ci peut contenir. Cela permet d'avoir des nombres plus petits et donne des quantités plus simples à manipuler.

Kilo, giga, et compagnie

Comme dit plus haut, nos mémoires n'ont pas toutes la même capacité : suivant la mémoire, elle peut varier de quelques octets à plusieurs milliards d'octets. Le seul problème, c'est que les mémoires actuellement présentes dans nos ordinateur sont tout de même assez grosses : cela se compte en millions ou milliards d'octets. Et je ne vous apprends rien en disant que manipuler des quantités dépassant le milliard est loin d'être facile. Pour se faciliter la tache, on utilise des préfixes pour désigner les différentes capacités mémoires. Vous connaissez surement ces préfixes : kibioctets, mebioctets et gibioctets, notés respectivement Kio, Mio et Gio.

Préfixe	Quantité	Puissance de deux
Kio	1024	2^{10} octets
Mio	1 048 576	2^{20} octets
Gio	1 073 741 824	2^{30} octets

Ainsi,

- un kibioctet correspond à 1024 octets ;
- un mébioctet correspond à 1 1 048 576 octets ;
- un gibioctet correspond à 1 073 741 824 octets.

De même,

- un kibibit correspond à 1024 bits ;
- un mébibit correspond à 1 1 048 576 bits ;
- un gibibit correspond à 1 073 741 824 bits.

Ainsi, un kibioctet vaut 1024 octets, un mébioctet en vaut 1024^2, un gibioctet vaut 1024^3 octets, etc.

Pourquoi utiliser des puissances de 1024, et ne pas utiliser des puissances un peu plus communes ?

Dans la majorité des situations, les électroniciens préfèrent manipuler des puissances de deux pour se faciliter la vie, et c'est aussi le cas pour les mémoires : il est plus simple de concevoir des mémoires qui contiennent un nombre de cases mémoires qui soit une puissance de deux. Par convention, on utilise souvent des puissances de 1024, qui est la puissance de deux la plus proche de 1000.

Bizarre, j'ai toujours entendu parler de kilo-octets, méga-octets, gigaoctets, etc. C'est normal ?

Et bien non ! Logiquement, on ne devrait pas parler de kilo-octets, méga-octets ou gigaoctets : c'est encore une fois un abus de langage.

Dans le langage courant, kilo, méga et giga sont des multiples de 1000. Quand vous vous pesez sur votre balance et que celle-ci vous indique 58 kilogrammes (désolé mesdames), cela veut dire que vous pesez 58000 grammes. De même, un kilomètre est égal à mille mètres, et non 1024 mètres.

Autrefois, on utilisait les termes kilo, méga et giga à la place de nos kibi, mebi et gibi, par abus de langage : les termes kibi, mébi et gibi n'existaient pas. Pour éviter les confusions, de nouvelles unités (les kibi, gibi et autres) ont fait leur apparition. Malheureusement, peu de personnes sont au courant de l'existence de ces nouvelles unités, et celles-ci sont rarement utilisées. Aussi, ne vous étonnez pas si vous entendez parler de gigaoctets en lieu et place de gibioctets : cette confusion est très courante.

Saviez-vous que cette confusion permet aux fabricants de disques durs de nous "arnaquer" ? Ceux-ci donnent la capacité des disques durs qu'ils vendent en kilo, mega ou giga octets. L'acheteur croit implicitement avoir une capacité exprimé en kibi, mebi ou gibi octets, et se retrouve avec un disque dur qui contient moins de mémoire que prévu. C'est pas grand chose de perdu, mais il faut avouer que c'est tout de même de l'arnaque !

Mémoires volatiles et non-volatiles

Vous avez déjà remarqué que lorsque vous éteignez votre ordinateur, le système d'exploitation et les programmes que vous avez installés...ne s'effacent pas. Par contre, certaines informations (comme le document Word que vous avez oublié de sauvegarder avant que votre PC plante) s'effacent dès le moment où l'ordinateur s'éteint.

Oui, et alors ? Quel est le rapport avec les mémoires ?

Très simple : vos programmes et le système d'exploitation sont placés sur une mémoire qui ne s'efface pas quand on coupe le courant, pas votre document Word. On dit que la mémoire dans laquelle votre OS et vos programmes étaient placés est une mémoire non-volatile, tandis que celle qui stockait votre document Word était une mémoire volatile.

Mémoires Non-volatiles	Mémoires Volatiles
Conservent leurs informations quand on coupe le courant	Perdent leurs informations lors d'une coupure de l'alimentation

Comme exemple de mémoire non-volatile, on peut citer le disque dur. Tous vos programmes et votre système d'exploitation sont stockés dessus. Et quand vous débranchez votre ordinateur, ils ne s'effacent pas. Notre disque dur est donc une mémoire non-volatile.

Au fait : Les mémoires volatiles ne volent pas et n'ont pas de plumes !

RWM ou ROM

Une autre différence concerne la façon dont on peut accéder aux informations stockées dans la mémoire. Cette autre différence classe les mémoires en mémoires RWM, et ROM.

Mémoires ROM	Mémoires RWM
On peut récupérer les informations dans la mémoire, mais pas les modifier : la mémoire est dite **accessible en lecture**	On peut récupérer les informations dans la mémoire et les modifier : la mémoire est dite **accessible en lecture et en écriture**

A l'heure actuelle, les mémoires non-volatiles présentes dans nos ordinateurs sont toutes des mémoires ROM, (sauf le disque dur qui est accessible en lecture et écriture).

Attention aux abus de langage : le terme mémoire RWM est souvent confondu dans le langage commun avec les mémoires RAM.

PROM

Néanmoins, il existe des mémoires ROM un peu spéciales : on ne peut pas accéder en écriture à une donnée bien précise et ne modifier que celle-ci, mais on peut réécrire intégralement son contenu. On dit qu'on reprogramme la mémoire, ce qui est différent d'une écriture. Ce terme de programmation vient du fait que les mémoires ROM sont souvent utilisées pour stocker des programmes sur certains ordinateurs assez simples : modifier le contenu de ces mémoires revient donc à modifier le programme contenu dans la mémoire et donc reprogrammer l'ordinateur.

Néanmoins, il faut bien comprendre la différence entre

- écrire dans une mémoire : je sélectionne une case mémoire et je modifie son contenu, mais je ne touche pas aux autres cases mémoires ;
- reprogrammer : on efface tout et on recommence !

Ces mémoires sont appelées des mémoires **PROM**. Il existe plusieurs versions de ces mémoires PROM, qui ont chacune leurs caractéristiques.

On peut par exemple mentionner les **FROM**, qui sont fournies intégralement vierges, et on peut les reprogrammer une seule et unique fois. Ces mémoires sont souvent fabriquées avec des diodes ou des transistors qui serviront à stocker un bit. La programmation d'une telle ROM est très simple : pour écrire un zéro, il suffit de faire claquer la diode ou le transistor correspondant au bit qu'on veut modifier ! Pour stocker un un, on laisse notre diode ou transistor indemne. Vu qu'une diode ou un transistor ne se réparent pas tout seuls, on ne pourra pas changer le bit enregistré : impossible de transformer un zéro en un : notre mémoire est programmée définitivement.

Viennent ensuite les **EPROM**, qui peuvent être effacées et reprogrammées plusieurs fois de suite sans problèmes, contrairement aux FROM. En effet, ces mémoires s'effacent lorsqu'on les soumet à des rayonnements UV : autant dire que l'effacement n'est pas

très rapide.

D'autres mémoires ROM peuvent être effacées par des moyens électriques : ces mémoires sont appelées des **mémoires EEPROM**. Pour donner des exemples de mémoires EEPROM, sachez que vous en avez surement une dans votre poche. Et oui, votre clé USB est fabriquée avec une mémoire qu'on appelle de la mémoire FLASH, qui est une sorte d'EEPROM.

Le temps d'accès

Imaginons que l'on souhaite accéder à une donnée localisée dans une mémoire. On peut vouloir la lire, voire l'écrire si c'est une mémoire RWM, peu importe. Que ce soit une lecture ou une écriture, il va falloir attendre un certain temps que notre mémoire aie finie de lire ou d'écrire notre donnée. Et ce temps, c'est ce qu'on appelle le **temps d'accès**.

Sur certaines mémoires, lire une donnée ne prend pas le même temps que l'écrire. On se retrouve alors avec deux temps d'accès : un temps l'accès en lecture et un temps d'accès en écriture. Généralement, la lecture est plus rapide que l'écriture. Il faut dire qu'il est beaucoup plus fréquent de lire dans une mémoire qu'y écrire, et les fabricants préfèrent donc diminuer au maximum le temps d'accès en lecture que toucher aux temps d'écriture.

Ce temps d'accès varie beaucoup suivant le type de mémoire. De plus, sur certaines mémoires, le temps d'accès dépend parfois de la position de la donnée en mémoire. C'est le cas sur les disques durs, par exemple, ou sur les mémoires à accès séquentiel.

Mémoires RAM

Les mémoires RAM sont des mémoires qui sont adressables. Mais en plus, les mémoires RAM ont une particularité : le temps d'accès est *toujours le même*, quelle que soit l'adresse de la donnée que l'on souhaite consulter ou modifier. Toutes les mémoires n'ont pas cette particularité : ce n'est pas le cas d'un disque dur, par exemple, dont le temps d'accès dépend de l'emplacement de l'information sur le disque dur et de la position de la tête de lecture.

Toutes les mémoires RAM actuelles sont des mémoires volatiles. Néanmoins, il existe des projets de recherche qui travaillent sur la conception d'une mémoire nommée la MRAM qui serait une mémoire RAM non-volatile. Reste à finir le travail de recherche, ce qui n'est pas pour tout de suite !

Il existe deux types de RAM : les **SRAM** ou ram statiques et les **DRAM** ou RAMs dynamiques

Les SRAM

Les données d'une SRAM ne s'effacent pas tant qu'elles sont alimentées en courant.

Ces mémoires sont souvent (bien que ce ne soit aps une obligation) fabriquées avec des bascules, ces fameux circuits de mémorisation qu'on a vu il y a de cela quelques chapitres. Vu que ces fameuses bascules utilise pas mal de transistors (au minimum 6, voire plus), nos cellules mémoires auront tendance à prendre un peu de place. En conséquence, une cellule mémoire de SRAM est plus grosse qu'une cellule de mémoire DRAM (qui n'utilise qu'un seul transistor, et un autre composant élèctronique : un condensateur), ce qui signifie qu'on peut mettre beaucoup moins de cellules de SRAM que de cellules de DRAM sur une surface donnée. Pour cette raison, on dit souvent que nos mémoires SRAM ne peuvent contenir beaucoup de bits. Cette mémoire SRAM est donc utilisée lorsque l'on souhaite avoir une mémoire rapide, mais en faible quantité.

Elle sont assez rapides, mais très chères. Pour info, votre processeur contient beaucoup de mémoires directement intégrées dans ses circuits (les registres et les caches) qui sont toutes faites avec de la SRAM. Il faut dire que les mémoires intégrées au processeur ont absolument besoin d'être rapides, et qu'on a rarement besoin d'en mettre beaucoup, ce qui fait que la SRAM est un choix assez adapté.

Les DRAM

Avec les DRAM, les bits stockés en mémoire s'effacent tout seul en quelques millièmes ou centièmes de secondes (même si l'on n'y touche pas). Et c'est sans compter que lire une donnée stockée en mémoire va obligatoirement effacer son contenu. Il faut donc réécrire chaque bit de la mémoire régulièrement, ou après chaque lecture, pour éviter qu'il ne s'efface. On appelle cela le **rafraîchissement mémoire**.

Ce rafraîchissement prend du temps, et a tendance à légèrement diminuer l'efficacité des DRAM. Autrefois, ce rafraichissement était effectué par un circuit placé sur la carte mère de notre ordinateur, qui était chargé de toute la gestion de la mémoire (on l'appelle le contrôleur mémoire). Dans une telle situation, les ordres de rafraîchissement de la mémoire transitent donc par le bus, le rendant temporairement incapable de transférer des données. Ce défaut, autrefois très pénalisant ne l'est plus de nos jours : de nos jours, les mémoires RAM contient un circuit qui se charge de rafraichir automatiquement les données présente dans notre mémoire DRAM. Les pertes de performances sont ainsi plus mitigées.

Les DRAM sont donc plus lentes que les SRAM, mais peuvent stocker beaucoup plus de bits pour une surface ou un prix identique. Il faut dire qu'une cellule mémoire de DRAM prend beaucoup moins de place qu'une cellule de SRAM, ce qui fait que les mémoires DRAM ont souvent une grande capacité comparé aux SRAM. C'est ce qui fait qu'elles sont utilisées pour la mémoire principale de nos PC : quand on lance plusieurs programmes assez gourmand en même temps en plus du système d'exploitation, il vaut mieux avoir suffisamment de RAM.

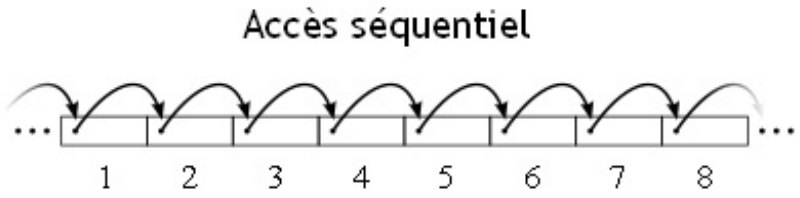
Donnée, où es-tu ?

Nos mémoires se différencient aussi par l'ordre dans lequel on peut accéder à leurs données.

Mémoires Séquentielles

Sur les anciennes mémoires, comme les bandes magnétiques, on était obligé d'accéder aux données dans un ordre prédéfini. On parcourait ainsi notre mémoire dans l'ordre, en commençant par la première donnée, et en passant à la donnée suivante après une lecture ou une écriture : c'est ce qu'on appelle l'**accès séquentiel**.

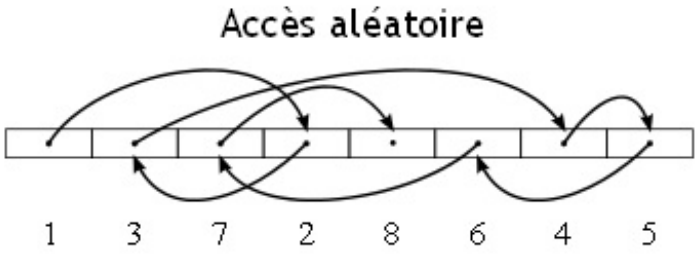
Pour lire ou écrire une donnée, il fallait visiter toutes les cases mémoires précédentes avant de tomber sur la donné recherchée. Et impossible de revenir en arrière ! Sauf à reprendre le lecture/écriture depuis le début de la mémoire. Des mémoires dans le genre se passent complètement d'adressage : on n'a pas le besoin, ni la possibilité de sélectionner une donnée dans la mémoire avec une adresse.



De nos jours, l'accès séquentiel est obsolète et presque complètement inutilisée : seules quelques vielles mémoires utilisaient ce genre d'accès.

Mémoires à accès aléatoire

Les mémoires actuelles utilisent plutôt ce qu'on appelle **l'accès aléatoire**. Avec cet accès aléatoire, on peut accéder à chaque case mémoire dans n'importe quel ordre, sans se soucier des données déjà parcourues avant ou de la position des données en mémoire.



On peut accéder à n'importe quelle donnée dans notre mémoire, sans trop se soucier de l'ordre d'accès. Pour accéder à une donnée, on est obligé d'indiquer à notre mémoire la position de celle-ci dans la mémoire.

Et on fait comment ?

On utilise toutes une méthode déjà vue auparavant : l'**adressage**.

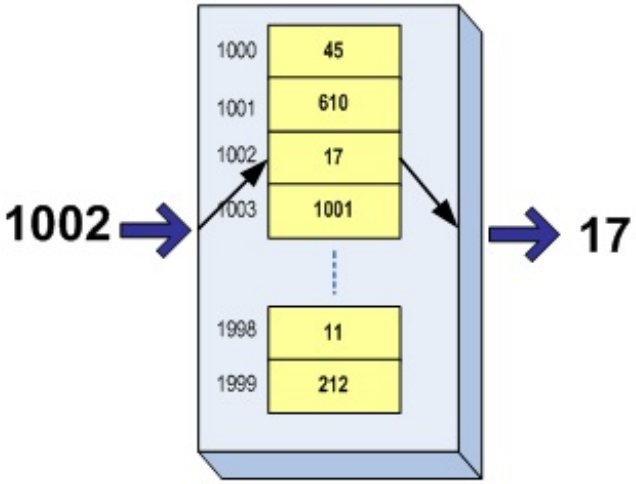
Cette solution est très simple : chaque case mémoire se voit attribuer un nombre binaire unique, **l'adresse**, qui va permettre de la sélectionner et de l'identifier celle-ci parmi toutes les autres.

Adresse	Contenu mémoire

0	11101010 01011010
1	01111111 01110010
2	00000000 01111100
3	01010101 0000000
4	10101010 00001111
5	00000000 11000011

En fait, on peut comparer une adresse à un numéro de téléphone (ou à une adresse d'appartement) : chacun de vos correspondants a un numéro de téléphone et vous savez que pour appeler telle personne, vous devez composer tel numéro. Ben les adresses mémoires, c'est pareil !

Exemple : on demande à notre mémoire de sélectionner la case mémoire d'adresse 1002 et on récupère son contenu (ici, 17).



Mémoires FIFO

Nous avons donc deux méthodes d'accès assez extrêmes : les mémoires à accès séquentielles, et les mémoires à accès aléatoires. Ceci dit, les mémoires à accès séquentielles ne sont pas les seules à imposer un ordre d'accès aux données. Il existe deux autres types de mémoire qui forcent l'ordre d'accès. Ce sont les mémoires FIFO et LIFO. Commençons par voir les mémoires FIFO.

Ces mémoires sont des mémoires dans lesquelles les données sont triées par ordre d'arrivée. Les données sont donc écrites dans la mémoire unes par unes, et placées dedans au fur et à mesure. Une lecture ne renverra que la donnée la plus ancienne présente dans cette mémoire. De plus, la lecture sera destructrice : une fois la donnée lue, elle est effacée.

On trouve ces mémoires à l'intérieur de nos processeurs : diverses structures matérielles sont conçues à partir de mémoires FIFO.

Il est facile de créer ce genre de mémoire à partir d'une mémoire RAM : il suffit juste de rajouter des circuits pour gérer les ajouts/retraits de données. On a notamment besoin de deux registres : un pour stocker la dernière donnée ajoutée, et un autre pour localiser la donnée la plus ancienne.

Mémoires LIFO

Poursuivons maintenant avec les mémoires LIFO. Ces mémoires sont des mémoires dans lesquelles les données sont triées par ordre d'arrivée. Les données sont donc écrites dans la mémoire unes par unes, et placées dedans au fur et à mesure. Une lecture ne renverra que la donnée la plus récente présente dans cette mémoire. De plus, la lecture sera destructrice : une fois la donnée lue, elle est effacée.

On peut voir ces mémoires LIFO comme des mémoires qui fonctionnent sur le même principe qu'une pile. En clair, toute écriture

empilera une donnée au sommet de cette mémoire LIFO. Toute lecture dépilera la donnée située au sommet de la mémoire LIFO. D'ailleurs, je tiens à signaler qu'il est facile de créer ce genre de mémoire à partir d'une mémoire RAM : il suffit juste de rajouter un registre qui stocke l'adresse du sommet de la pile, ainsi que quelques circuits pour gérer les empilements/dépilements.

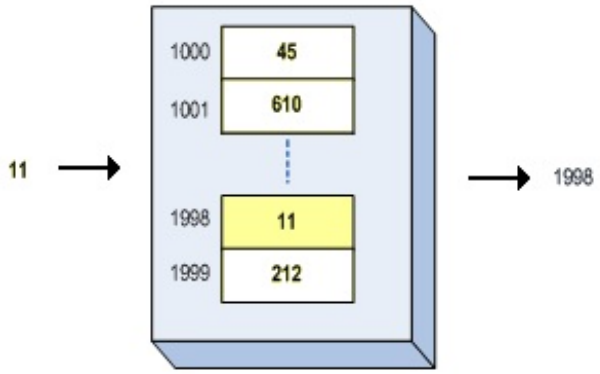
On trouve ces mémoires à l'intérieur de nos processeurs : diverses structures matérielles sont conçues à partir de mémoires FIFO.

Content Adressables Memories

Enfin, on trouve un dernier mode d'accès : l'accès par contenu. Il existe des mémoires assez spéciale, nommées les Content Adressables Memories, ou encore mémoire adressables par contenu, qui implémente ce mode d'accès. Ces mémoires sont de deux types.

Adress Return

Tout d'abord, il existe un premier type de mémoires : les mémoires Adress Return. Sur ces mémoires, on fait à peu-prêt la même chose qu'avec une mémoire à accès aléatoire, mais dans le sens inverse. Au lieu d'envoyer l'adresse pour accéder à la donnée, on va envoyer la donnée pour récupérer son adresse.



Cela peut paraitre bizarre, mais ces mémoires sont assez utiles dans certains cas de haute volée. Dès que l'on a besoin de rechercher rapidement des informations dans un ensemble de donnée, ou de savoir si une donnée est présente dans un ensemble, ces mémoires sont reines. Certaines circuits internes au processeur ont besoin de mémoires qui fonctionnent sur ce principe. Mais laissons cela à plus tard.

Pour ceux qui sont intéressés, sachez que j'ai écrit un tutoriel sur le sujet, disponible ici : Les mémoires associatives. Je vous conseille de lire celui-ci une fois que vous aurez terminé de lire ce tutoriel.

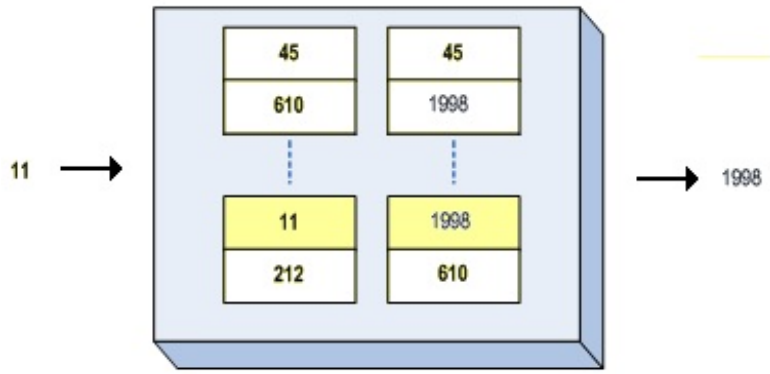
Hash table

Ensuite, on trouve un deuxième type de mémoire adressable par contenu : les mémoires à correspondance. Sur ces mémoires, chaque donnée se voit attribuer un identifiant, qu'on appelle le Tag. Une mémoire à correspondance stocke des couples Tag / Donnée. Il est possible que plusieurs données différentes aient le même Tag, ou réciproquement, que des données identiques aient des Tags différents.

Key-value pairs (records)

Tag	Data
Tag	Data
Tag	Data
Tag	Data
Tag	Data
Tag	Data

A chaque accès mémoire, on envoie le Tag de la donnée à modifier, et la mémoire accède alors directement à la donnée.



Ce principe est très utilisé dans nos ordinateurs. Certaines mémoires intégrées au processeur utilisent ce mode d'accès. On utilise aussi ce genre de mémoire dans des applications utilisant des bases de données ou pour simuler des réseaux de neurones.

Une histoire de bus

Une mémoire est reliée au reste de l'ordinateur via trois bus :

- le bus de données, qui transporte les données lue ou à écrire en mémoire ;
- un bus de commande, qui permet d'envoyer des ordres à la mémoire ;
- et éventuellement un bus d'adresse pour les mémoires à accès aléatoire.

Dans ce qui va suivre, nous allons voir en détail ces différents bus.

Bus de commande

Commençons par le bus de commandes. Dans sa version minimale, il sert à indiquer les sens de transferts des données à la mémoire : s'agit-il d'une écriture, d'une lecture, etc. Pour les mémoires ROM, on sait d'avance que le composant qui va adresser la mémoire ne peut que faire une lecture : il n'y a pas besoin de préciser que c'est une lecture. Il est donc parfois possible de s'en passer.

Mais pour les mémoires RWM, c'est autre chose : on peut aussi bien y accéder en écriture qu'en lecture. On peut accéder à une adresse de deux façons :

- soit on enregistre une information dans la mémoire : c'est une **écriture**.
- soit on récupère une information stockée dans la mémoire : c'est une **lecture**.

Pour préciser le sens de transfert à la mémoire, on utilise un bit du bus de commande nommé **R/W**. Il est souvent admis par convention que R/W à 1 correspond à une lecture, tandis que R/W vaut 0 pour les écritures.

Bus d'adresse

Pour choisir la case mémoire à laquelle on veut accéder, il faut bien pouvoir spécifier son adresse à notre mémoire. Pour cela, notre mémoire contient des entrées sur laquelle on peut placer notre adresse mémoire, reliés au reste du circuit par des fils. Les fils du bus qui transmettent l'adresse vers la mémoire sont regroupés dans une sorte de "sous-bus" qu'on appelle le **bus d'adresses**.

Memory Map

Ainsi, quelle que soit la case mémoire à laquelle on souhaite accéder, il suffit d'envoyer son adresse sur le bus d'adresse et la mémoire sélectionnera cette case pour nous. Peu importe que cette case mémoire soit en RAM, un registre, ou un registre qui permet la communication avec les périphériques. En effet, sur certains ordinateurs, on utilise un seul bus d'adresse pour gérer plusieurs mémoires différentes : certaines adresses sont attribuées à la mémoire RAM, d'autres à la mémoire ROM, d'autres à des périphériques, etc.

Par exemple, certains périphériques possèdent des mémoires internes. Ces mémoires internes possèdent parfois des adresses, et sont donc adressables directement par le processeur. Ce qui fait, par exemple, que la mémoire de votre carte graphique se retrouve dans l'espace d'adressage du processeur.Et oui, votre processeur peut s'occuper d'une partie de la gestion de la mémoire de votre carte graphique.

Il existe une table pour chaque ordinateur qui définit à quels composants électroniques sont attribuées les adresses. C'est la ***memory map***.

Voici un exemple :

Adresse	Composant élèctronique
de 0000 0000 à 0000 0011	Registres du processeur
de 0000 0011 à 0000 1111	Mémoire Programme
de 0000 1111 à 0011 1111	Mémoire RAM
de 0011 1111 à 0110 0000	Registres De communication Avec le périphérique 1
de 0110 0000 à 1111 1111	Mémoire du périphérique 1

La gestion de l'adressage (et donc du contenu du bus d'adresse) est réalisée dans la majorité des cas par le processeur, qui possède des unités permettant de gérer les bus de commande et d'adresse. Mais il arrive que les périphériques puissent accéder directement à la mémoire : il suffit que ces périphériques et le processeur de l'ordinateur utilisent une technologie nommée *Direct Memory Acces*. On en reparlera plus tard dans la suite de ce tutoriel.

Une histoire de capacité

Évidemment, plus on utilise une mémoire ayant une capacité importante, plus on devra utiliser un grand nombre d'adresses différentes : une par case mémoire. Or, une adresse est représenté dans notre ordinateur par un nombre strictement positif, codé en binaire. Si on utilise n bits pour représenter notre adresse, alors celle-ci peut prendre 2^n valeurs différentes, ce qui fait 2^n cases mémoires adressables. C'est pour cela que nos mémoires ont toujours une capacité qui est une puissance de deux !

Au fait, vous vous souvenez que je vous ai dit que dans la majorité des ordinateurs actuels, chaque case mémoire pouvait stocker 8 bits ?

Et bien voilà l'explication : en utilisant 8 bits par case mémoire, on utilise 8 fois moins d'adresse qu'en utilisant des cases mémoires de 1 bit. Cela diminue donc le nombre de fil à câbler sur le bus d'adresse.

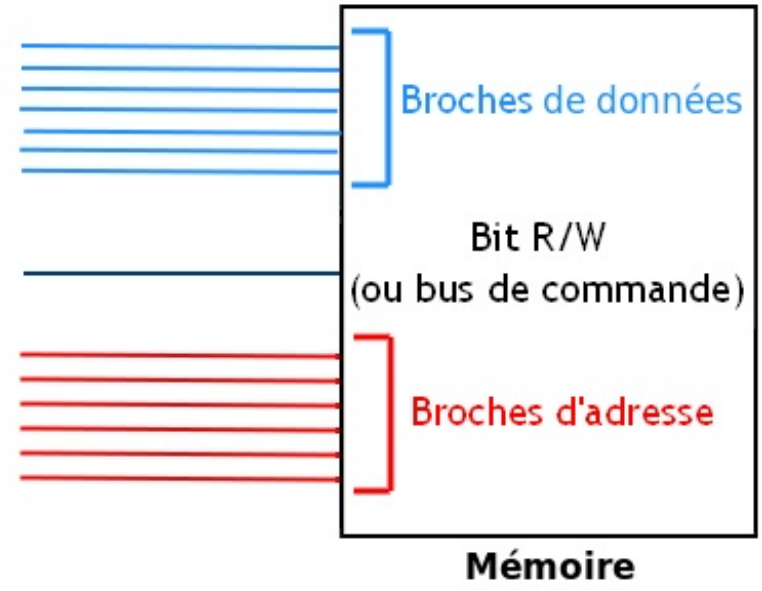
Mais attention : toutes les mémoires n'ont pas des cases mémoires d'une taille de 8 bits. Si vous regardez les anciens ordinateurs, vous verrez qu'autrefois, nos mémoires utilisaient des cases mémoires plus petites, contenant 2, 3, 4, 7 bits. Il est même arrivé que certaines mémoires soient *bit-adressables*, c'est à dire qu'on pouvait adresser chaque bit individuellement. De même, rien

n'empêche d'aller au delà de 8 bits : certains ordinateurs ont, ou avaient, des cases mémoires de 16 ou 18 bits. Mais pour être franc, ce genre de chose est assez rare de nos jours.

Connexion du bus sur la mémoire

Bus de donnée, bus d'adresse, bit R/W : tout cela doit être relié à la mémoire. Pour cela, notre mémoire possède des **broches**, qui sont des morceaux de métal ou de conducteur sur lesquelles on va venir connecter nos bus. Cela permettra à notre mémoire de pouvoir communiquer avec l'extérieur.

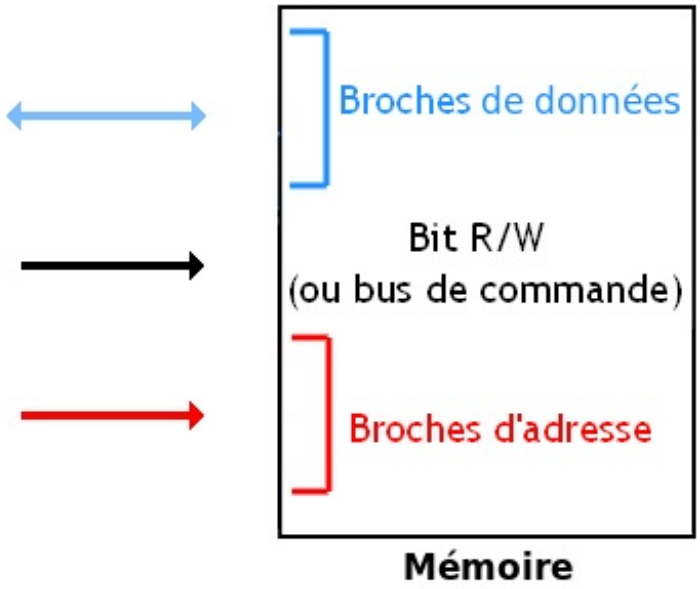
Dans le cas le plus simple, une mémoire est connectée au bus comme ceci :



La mémoire contient donc des broches sur lesquels brancher nos bus, avec des broches réservées au bus d'adresse, d'autres réservées au bus de donnée, et une réservée au bit R/W.

On remarque que les informations présentes sur le bus d'adresse et sur le fil R/W vont dans un seul sens : du bus vers la mémoire, mais pas l'inverse. Une mémoire n'a pas à envoyer une adresse sur ce bus, ou à demander une lecture/écriture à qui que ce soit : les bits qui se trouvent sur ces broches serviront à commander la mémoire, mais ne pourra pas être modifié par celle-ci. Nos broches sur lesquelles on connecte nos bus d'adresse et notre bit R/W sont donc des **entrées** : ce qui se trouve dessus va renter dans le composant qui fera ce qu'il faut avec, mais le composant ne pourra pas modifier le contenu de ces broches.

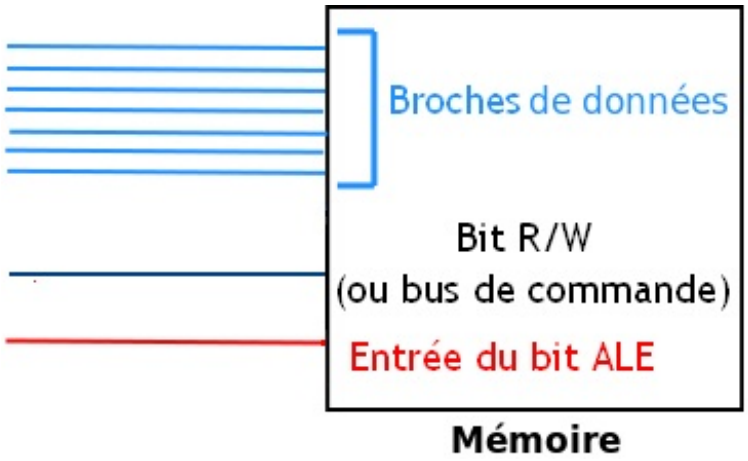
Pour les broches reliées au bus de donnée, le cas est plus litigieux et dépend de l'accès effectué. Si c'est une écriture, ces broches seront des entrées : la donnée sera lue par la mémoire sur ces entrées. Dans le cas d'une lecture,la mémoire va modifier le contenu du bus de donnée, et ces broches vont donc se comporter comme des **sorties**.



Bus multiplexé

Néanmoins, on peut remarquer que cela fait beaucoup de fils et beaucoup de broches. Cela peut poser problème : les électroniciens qui conçoivent ce genre de circuits essayent au maximum de limiter le nombre de fils pour éviter les problèmes. Il y a des raisons à cela : un processeur possède lui aussi des entrées et des sorties. Parfois, on peut vouloir câbler un grand nombre de composants dessus : on utilise alors beaucoup d'entrées et il n'en reste plus pour câbler un bus complet dessus. Sans compter le bordel pour câbler un grand nombre de fils sur une carte d'une taille limitée.

Il existe alors une petite astuce pour économiser des fils : utiliser un seul bus qui servira alternativement de bus de donnée ou d'adresse. Cela s'appelle multiplexer le bus d'adresse et de donnée. Voici à quoi ressemble donc l'ensemble bus-mémoire :



On a donc :

- un seul bus qui sert successivement de bus de données et de bus mémoire ;
- un bus de commande, avec éventuellement un bits R/W ;
- un bit ALE : celui-ci vaut 1 quand une adresse transite sur le bus, et 0 si le bus contient une donnée (ou l'inverse!).

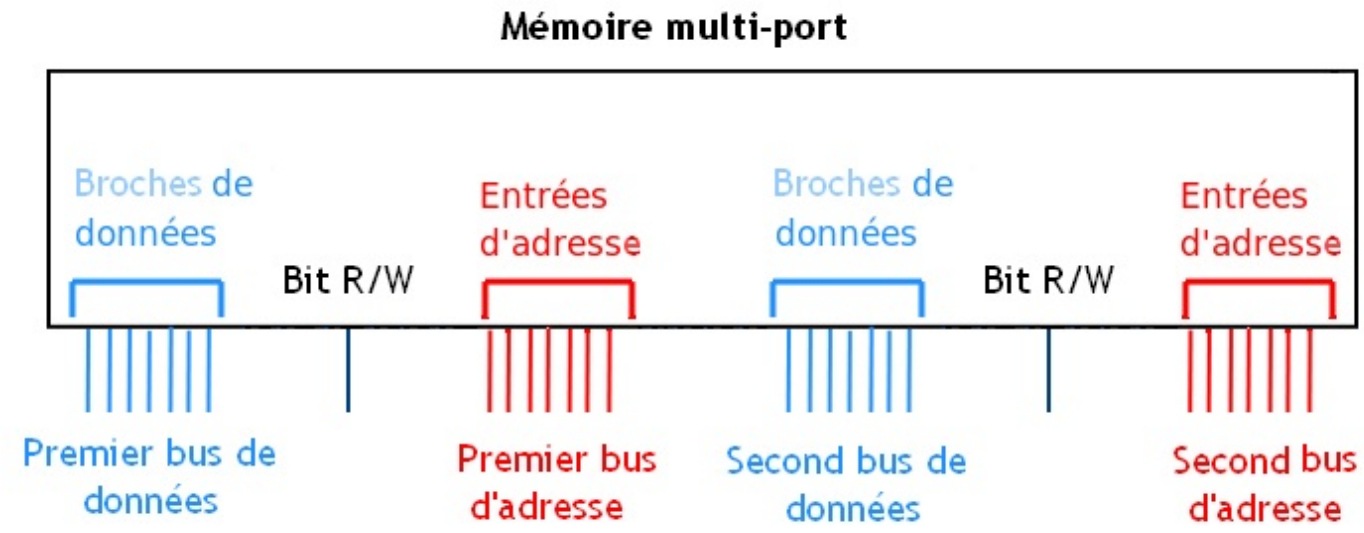
Ce genre de bus est plus lent qu'un bus qui ne serait pas multiplexé lors des écritures : lors d'une écriture, on doit en effet envoyer en même temps l'adresse et la donnée à écrire. Avec un bus multiplexé, on ne peut pas envoyer à la fois l'adresse, et une donnée (pour une écriture, par exemple). Cela doit être fait en deux passes : on envoie l'adresse d'abord, puis la donnée ensuite. Sur un bus qui n'est pas multiplexé, on peut envoyer l'adresse et la donnée en une seule étape, ce qui est plus rapide que de le faire en deux étapes.

Par contre, les lectures ne posent pas de problèmes : quelque soit le type de bus utilisé, on envoie d'abord l'adresse, et on récupère la donnée lue après l'envoi de l'adresse. Vu que ces deux opérations ne se font pas en même temps et qu'il n'y a pas

besoin d'envoyer à la fois une adresse et une donnée sur le bus, le bus multiplexé ne pose pas de problèmes. Heureusement, les lectures en mémoire sont bien plus courantes que les écritures, ce qui fait que la perte de performance due à l'utilisation d'un bus multiplexé est souvent supportable.

Mémoire multiports

Après avoir vu des mémoires cherchant à limiter le nombre de fils en payant le prix en terme de performances, je suis obligé de mentionner le cas inverse : des mémoires qui n'hésitent pas à câbler un maximum de fils pour gagner en performances. Ces mémoires possèdent plusieurs bus de données, d'adresse et de commande : chaque bus est présent en deux, trois ou quatre exemplaires. Ainsi, on pourra relier la mémoire à plusieurs bus, qui permettront de transférer chacun une donnée. Ces mémoires sont appelées des **mémoires multiports**. Ce nom vient simplement du fait que chaque bus (je parle d'un bus complet, avec bus d'adresse + bus de donnée + bus de commande) s'appelle un **port**, et que ces mémoires en ont plusieurs.



Cela permet de transférer plusieurs données en une seule fois : une sur chaque bus. On peut ainsi sélectionner plusieurs cases mémoires, et transférer des données avec chacune de ces cases simultanément (via chaque bus de donnée). Je suppose que vous voyez à quel point cela peut être plus rapide que d'envoyer chacune de ces donnée en une étape.

Certaines mémoires multiports possèdent des bus spécialisés pour l'écriture ou le lecture. Par exemple, certaines mémoires multiports peuvent être reliées à seulement deux bus : un sur lequel on ne peut que lire une donnée, et un autre sur lequel on ne peut qu'écrire. Les registres de nos processeurs sont de ce type : cela permet de simplifier la conception de notre processeur.

Ces bus pourront être reliés à des composants différents, ce qui fait que plusieurs composants pourront accéder en même temps à la mémoire. On peut aussi décider de relier la mémoire avec un seul composant, en utilisant tous les bus : le composant pourra alors modifier ou lire le contenu de tous les bus en même temps. De quoi effectuer plusieurs lectures/écritures en même temps.

Évidemment, cela fait énormément de fils à câbler, vu que certains bus sont en double, triple ou quadruple. Cela a un cout en terme de prix, mais aussi en terme de consommation énergétique : plus une mémoire a de ports, plus elle chauffe et consomme de l'électricité. Mais on peut gagner énormément en performances en utilisant de telles mémoires. Pour donner un exemple d'utilisation, les mémoires multi-ports sont utilisées dans les cartes graphiques actuelles, et pour fabriquer les registres du processeur.

Toutes les mémoires ne se valent pas !

Et oui, les mémoires ne se valent pas! Rien de méchant, rassurez-vous, c'est juste qu'il existe pleins de type de mémoires, avec leurs qualités et leurs défauts : certaines mémoires ont une plus grande capacité, une vitesse plus grande... Par contre un défaut revient quelque soit le type de mémoire : plus une mémoire peut contenir de données, plus elle est lente ! Sur ce point, pas de jaloux ! Toutes les mémoires sont égales, un vrai truc de soviet !

Une histoire de vitesse

On a vu au premier chapitre ce qu'était le temps d'accès d'une mémoire. Et bien il faut savoir que pour un type de mémoire (SRAM, DRAM, ROM...), le temps d'accès d'une mémoire dépend de sa capacité. Plus la capacité est importante, plus le temps d'accès est long. En clair, plus une mémoire est grosse, plus elle sera lente.

Pourquoi ?

La raison à cela est très simple : plus une RAM a une capacité importante, plus elle est grosse. Et plus elle est grosse, plus elle contient de portes logiques et plus les fils qui relient les divers composants de notre mémoire seront longs. Hors, le temps que met un signal électrique (un bit, quoi) pour aller d'un point à un autre du circuit gène la montée en fréquence. Ce temps s'appelle le temps de propagation, et on en a déjà parlé dans les chapitres au début de ce tutoriel. Ce temps de propagation dépend de pas mal de facteurs, dont le nombre maximal de portes que notre signal doit traverser (le *Critical Path*), ainsi que de la longueur des fils. Le fait est que plus la mémoire est grosse, plus ce temps de propagation est long. Un temps de propagation trop long aura des effets pas très reluisants : la fréquence de la mémoire sera faible, et notre mémoire sera lente. Alors certes, la capacité d'une mémoire ne fait pas tout et d'autres paramètres entrent en jeu, mais on ne peut pas passer ce problème sous le tapis.

Hiérarchie mémoire

Le fait est que si l'on souhaitait utiliser une seule grosse mémoire dans notre ordinateur, celle-ci serait donc fatalement très lente. Malheureusement, un composant très rapide ne peut attendre durant plusieurs millisecondes que la donnée soit chargée de la mémoire sans rien faire, ce serait gaspiller beaucoup trop de temps de calcul. On ne peut donc utiliser une seule grosse mémoire capable de stocker toutes les données voulues. Ce problème s'est posée dès les début de l'informatique. Les inventeurs des premiers ordinateurs modernes furent rapidement confrontés à ce problème.

Pour ceux qui ne me croient pas, regardez un peu cette citation des années 1940, provenant d'un rapport de recherche portant sur un des premiers ordinateurs existant au monde :

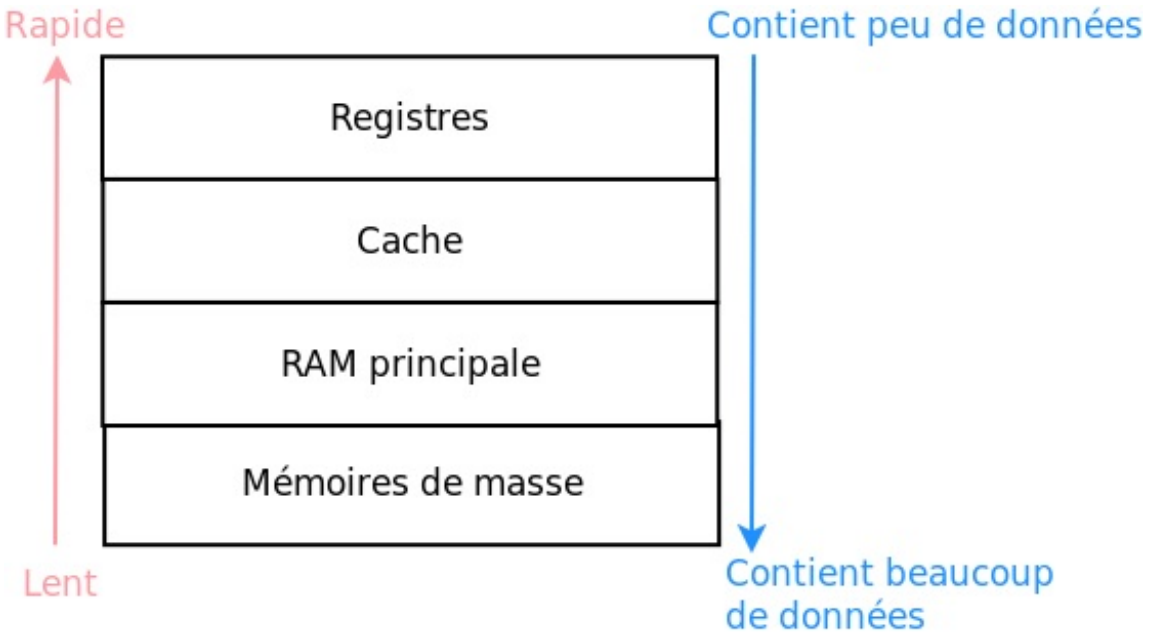
Citation : Burks, Goldstine, et Von Neumann

> Idéalement, nous désirerions une mémoire d'une capacité indéfiniment large tel que n'importe quel *byte* soit immédiatement accessible. Nous sommes forcés de reconnaitre la possibilité de la construction d'une hiérarchie de mémoire, chacune ayant une capacité plus importante que la précédente, mais accessible moins rapidement.

Comme on le voit, cette citation (traduite de l'anglais), montre le problème, mais évoque aussi la solution adoptée face à ce problème. Pour résoudre ce problème, il suffit de **segmenter la mémoire de l'ordinateur en plusieurs sous-mémoires, de taille et de vitesse différentes qu'on utilise suivant les besoins**. On aura donc des mémoires pouvant contenir peu de données dans lesquelles on pourra lire et écrire rapidement et des mémoires plus importante, mais plus lentes. Cette solution a été la première solution inventée pour résoudre ce problème et est encore massivement utilisée à l'heure actuelle : on n'a pas encore fait mieux !

Généralement, un ordinateur contient plusieurs mémoires de taille et de vitesse différentes.
Ces mémoires peuvent être classées en quatre grands types :

- **Les mémoires de masse** qui stockent des informations qui doivent être conservées même après extinction du système et qui sont accédée très rarement.
- **La mémoire principale** qui stocke toutes les informations temporaires auxquelles le processeur doit accéder peu souvent mais qui doivent être conservées suffisamment longtemps.
- **les mémoires caches**, qui accélèrent l'accès à la mémoire principale.
- **les registres**, très rapides contenant des instructions ou données que le processeur doit manipuler.



Localité de référence

On voit bien que les mémoires d'un ordinateur sont organisées de la plus lente à la moins lente. Le but de cette organisation est de placer les données accédées souvent, ou qui ont de bonnes chances d'être accédées dans le futur, dans une mémoire qui soit la plus rapide possible. Le tout est faire en sorte de placer les données intelligemment, et les répartir correctement dans cette hiérarchie des mémoires.

Ce placement se base sur deux principes qu'on appelle les principe de localité spatiale et temporelle. Pour simplifier :

- un programme a tendance à réutiliser les instructions et données qui ont étés accédées dans le passé : c'est la **localité temporelle** ;
- et un programme qui s'exécute sur un processeur à tendance à utiliser des instructions et des données qui ont des adresses mémoires très proches, c'est la **localité spatiale**.

Ces deux principes semblent très simples, mais sont lourds de conséquence. On peut exploiter ces deux principes pour placer correctement nos données dans la bonne mémoire. Par exemple, si on a accédée à une donnée récemment, il vaut mieux la copier dans une mémoire plus rapide, histoire d'y accéder rapidement les prochaines fois : on profite de la localité temporelle. On peut ainsi placer des données consultées ou modifiées fréquemment dans les registres ou la mémoire cache au lieu de les laisser en mémoire RAM. On peut aussi profiter de la localité spatiale : si on accède à une donnée, autant précharger aussi les données juste à coté, au cas où elles seraient accédées.

Placer les bonnes données au bon endroit (dans le cache plutôt qu'en RAM) permet d'avoir de sacrés gains de performances. Ce placement des données dans la bonne mémoire peut être géré par le matériel de notre ordinateur, par la façon dont sont construits nos programmes, ou gérable par le programmeur.
Par exemple :

- la mémoire cache est souvent gérée directement par le matériel de notre ordinateur, d'une façon qui peut être prise en compte par le programmeur ;
- les registres sont gérés par le programmeur (s'il programme ne assembleur ou en langage machine), ou par son langage de programmation (par le compilateur pour être précis) ;
- la RAM est implicitement gérée par le programmeur, etc.

Bref, sachez qu'un programmeur peut parfaitement prendre en compte le fait que les mémoires d'un ordinateur ne vont pas à la même vitesse, et peut concevoir ses programmes de façon à placer un maximum de données utiles dans la bonne mémoire. En effet, la façon dont est conçue un programme joue énormément sur la façon dont celui-ci accédera à ses données, et sur sa localité spatiale et temporelle. Vu que de nos jours, nos programmes passent de plus en plus de temps à attendre que les données à manipuler soient lues ou écrites depuis la mémoire, ce genre de choses commence à devenir une nécessité. Bref, un programmeur peut, et doit, prendre en compte les principes de localités vus plus haut dès la conception de ses programmes. Et cette contrainte va se faire de plus en plus forte quand on devra passer aux architectures multicœurs.

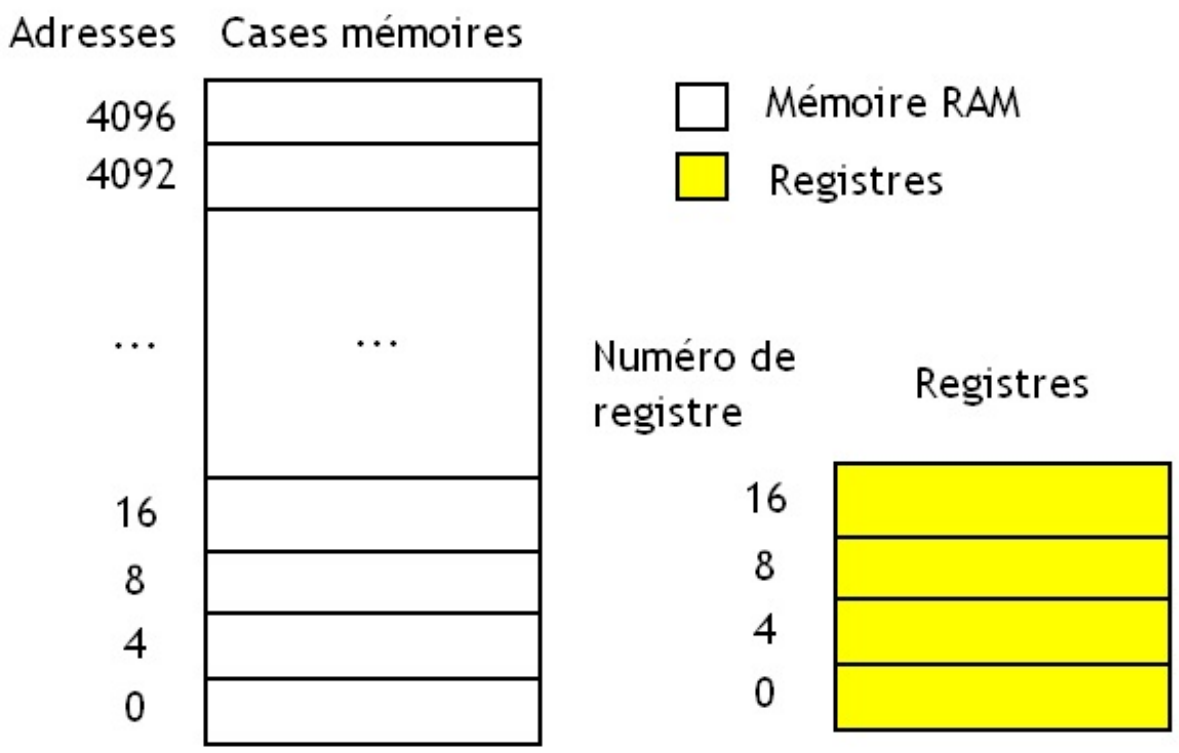
Maintenant que cette petite remarque est faite, parlons un peu des différentes mémoires. Commençons par ces fameux registres.

Registres

Les registres sont fabriqués avec des mémoire SRAM. Ces registres sont très souvent intégrés dans le processeur de votre ordinateur, mais quelques périphériques possèdent eux aussi des registres. La différence, c'est que les registres de notre processeur vont stocker temporairement des données pour pouvoir les manipuler rapidement. Les périphériques utilisent des registres pour communiquer avec le processeur, bien plus rapide qu'eux.

Noms de registres

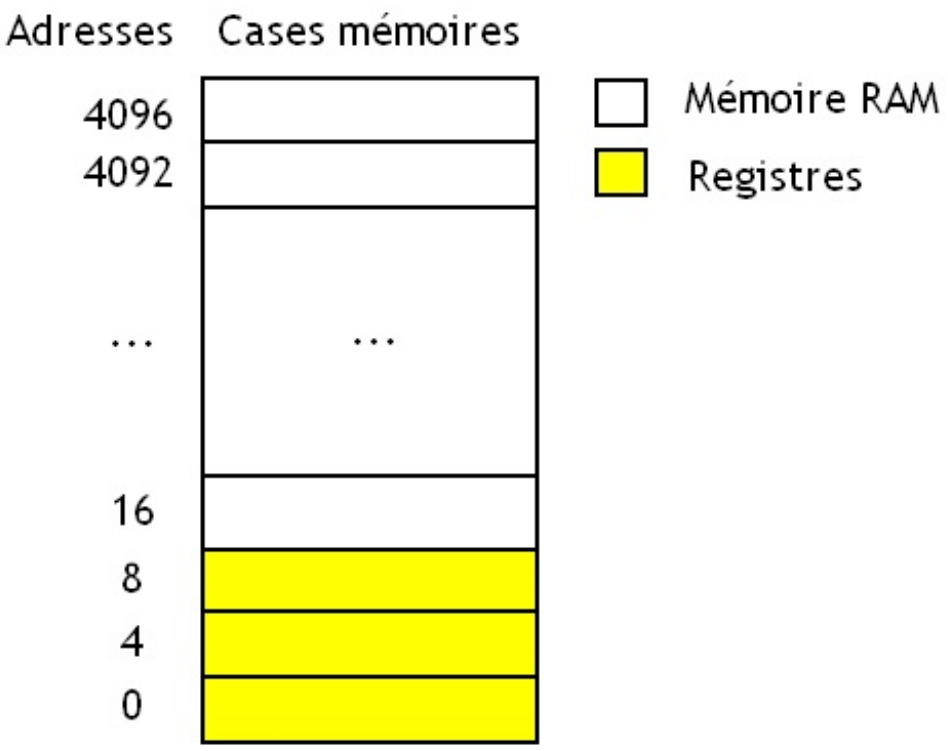
Sur les processeurs x86 ainsi que sur la grosse majorité des processeurs existants ou ayant existé, les registres ne sont pas adressables. Sur de tels processeurs, chaque registre est identifié par un numéro qui n'a rien à voir avec une adresse ! Ce numéro, ou nom de registre, permet d'identifier le registre que l'on veut, mais ne sort jamais du processeur : ce nom de registre, ce numéro, ne se retrouve jamais sur le bus d'adresse.



Quand une instruction voudra manipuler des données, elle devra fatalement donner leur position dans la mémoire. Pour manipuler une case mémoire, elle donne généralement son adresse, et pour un registre, elle donnera un nom de registre. Ces adresses et noms de registres seront codés sous la forme de suites de bits, incorporées dans l'instruction. Mais rien ne ressemble plus à une suite de bits qu'une autre suite de bits : notre processeur devra éviter de confondre suite de bits représentant une adresse, et suite de bits représentant un nom de registre. Pour éviter les confusions, chaque instruction devra préciser à quoi correspondra la suite de bits précisant la localisation des données à manipuler. On peut ainsi utiliser diverses instructions différentes suivant qu'on veut manipuler des registres ou des adresses mémoires, par exemple.

Registres adressables

Mais il existe quelques processeurs sur lesquels on peut adresser les registres via une adresse mémoire. Il est vrai que c'est assez rare, et qu'à part quelques vielles architectures ou quelques micro-contrôleurs, je n'ai pas d'exemples à donner. Mais c'est tout à fait possible ! C'est le cas du PDP-10.



Cache

Un cache est une mémoire qui doit être assez rapide, et est donc fabriqué avec de la SRAM, assez véloce. Ce cache est généralement intégré dans le processeur de votre ordinateur, mais quelques périphériques possèdent eux aussi des caches, comme certains disques durs.

Un cache n'est jamais adressable ! Cela est du au fait que chaque donnée présente dans la mémoire cache est une copie d'un emplacement de la mémoire RAM. Le contenu du cache est géré par un circuit particulier qui décide quoi charger dedans et quand. Lorsque le processeur veut ainsi accéder à une case mémoire en RAM (en lecture ou en écriture), il va envoyer l'adresse de cette case sur le bus. Celle-ci sera interceptée par les circuits chargés de gérer le cache qui regarderont alors si le cache contient une copie de la case à manipuler. Si c'est le cas, on lit ou écrit la donnée dans le cache. Dans le cas contraire, on accède à la mémoire RAM. Cela explique qu'on aie pas besoin de donner des adresses aux cases mémoires du cache : les circuits de gestion du cache savent à quelle case mémoire en RAM correspond chaque case mémoire du cache.

Local Stores

Sur certains processeurs, les mémoires caches sont remplacées par des mémoires qui fonctionnent différemment mais remplissent le même rôle : fournir un intermédiaire plus rapide entre les registres et la mémoire principale. A la place de mémoire cache, on utilise à la place ce qu'on appelle un ***Local Store***. Ce sont des mémoires RAM, identique à la méoire RAM principale, sauf que nos *Local Stores* sont plus petites et donc plus rapides. Contrairement aux mémoires caches, il s'agit de mémoires **adressables** ! Et cela change tout : vu que ces *Local Store* sont adressables, ils ne sont plus gérés automatiquement par le processeur. Ainsi, rien n'empêche le programmeur de décider quoi placer dans cette mémoire et quand : il peut s'en servir de mémoires tampon pour stocker des données qui seront réutilisées assez souvent et dans un intervalle de temps assez proche, sans avoir à stocker ces données en RAM.

Les transferts de données entre *Local Store* et mémoire RAM sont effectués par un circuit spécial, séparé du processeur. C'est le processeur qui configure ce circuit pour que celui-ci effectue le transfert désiré automatiquement, sans intervention du processeur. Dans ce genre de cas, la gestion d'un *Local Store* pose les mêmes problèmes que la gestion d'une mémoire cache : il faut notamment prendre en compte la localité spatiale et temporelle. Dans de telles conditions, on peut voir notre *Local Store* comme une sorte de mémoire cache gérée par le programmeur, qui décide quand lancer le transfert, quoi transférer, et où.

Avantages/inconvénients

Ces *Local Stores* consomment moins d'énergie que les caches à taille équivalente. En effet, ceux-ci sont de simples mémoires RAM, et on n'a pas besoin de circuits compliqués pour les gérer automatiquement. Ces circuits gérant le cache prennent beaucoup de place sur le processeur et ont un certain temps de latence. Les *Local Store* n'ont pas ces problèmes et peuvent être très avantageux.

Ces *Local Stores* peuvent aussi être très avantageux quand il s'agit de partager des données entre plusieurs processeurs efficacement. Dans ce genre de cas, l'utilisation de toute une hiérarchie de mémoires caches L1, L2, L3, etc ; pose des problèmes assez conséquents dont je ne parlerais pas ici (Ces fameuses histoires de cohérence des caches évoquée dans le chapitre précédent), qui peuvent fortement diminuer les performances. Les *Local Stores*, eux, ne posent presque aucun problème, et sont donc mieux adaptés à ce genre de situations.

Coté inconvénients, ces *Local Stores* peuvent entrainer des problèmes de compatibilité : que faire si jamais on souhaite changer leur taille ? On est obligé de changer certains programmes pour que ceux-ci puissent profiter de local stores plus grands, ou simplement pour que ceux-ci s'adaptent à une organisation de la mémoire un peu différente. Au final, on utilise ces *Local Stores* dans des situations pour lesquels on se moque de la compatibilité et pour lesquelles on veut un ordinateur qui chauffe peu et consomme assez peu d'énergie.

C'est pas si rare !

L'utilisation de *Local Stores* est tout de même quelque chose d'assez répandu. Pour donner quelques exemples, voici quelques ordinateur et processeurs assez connus utilisant un *Local Store* :

- le processeur SuperH, utilisé dans les consoles Sega Saturn, Sega 32X., ainsi que dans la Dreamcast ;
- les processeurs R3000 qu'on trouve dans la playstation 1;
- c'est aussi le cas de l'Emotion Engine, le processeur de la Playstation2 ;
- et tant qu'à parler de consoles de jeux, ne parlons pas des processeurs SPE intégrés dans le processeur de la Playstation 3, qui possèdent un Local Store de 256 kibioctets ;
- mais c'est sans oublier les cartes graphiques récentes pouvant utiliser CUDA : la geforce 8800 utilise un *Local Store* de 16 kibioctets, et les modèles récents ont un *Local Store* encore plus gros.

Mémoires principales

La mémoire principale sert de mémoire de travail, et parfois de mémoire programme. C'est dedans qu'on va stocker les données à manipuler et éventuellement le ou les programmes à exécuter (sur les architectures Von Neumann ou sur les architectures Harvard modifiées). Par conséquent, on va devoir accéder à son contenu assez souvent et pouvoir gérer cela de façon manuelle : toutes les mémoires RAM sont donc adressables.

La mémoire principale est fabriquée avec de la DRAM, qui peut contenir plus de données sur une surface égale que la SRAM. Il faut dire que contenir un système d'exploitation moderne et plusieurs programmes en même temps nécessite au moins quelques centaines de méga-octets. L'utilisation de DRAM au lieu de SRAM permet à notre mémoire d'atteindre les 2 à 4 gigaoctets sans frémir, ce qui fait tout de même 4 milliards de cases mémoires adressables pour 4 gigaoctets.

Néanmoins, il existe des exceptions qui confirment la règle : la gamecube et la Wii possèdent toutes les deux une mémoire principale de 24 mégas-octets de SRAM. Il s'agit toutefois d'une mémoire SRAM, améliorée de façon à augmenter sa densité (le nombre de *Bytes* qu'on peut placer sur une surface donnée).

Mémoires de masse

Ces mémoires servent surtout à stocker de façon permanente des données ou des programmes qui ne doivent pas être effacés : ce seront donc des mémoires non-volatiles, souvent fabriquées avec de la ROM ou des dispositifs magnétiques. Les mémoires de masse servent toujours à stocker un programme ou quelques paramètres/constantes utiles. On doit donc accéder à leur contenu et elles sont donc adressables, à part certaines vielles mémoires magnétiques qui font exception. Vu que ces mémoires doivent souvent stoker une grande quantité de bits (un programme prend souvent beaucoup de place), elles doivent avoir une capacité énorme comparé aux autres types de mémoires, et sont donc très lentes.

Parmi ces mémoires de masse, on trouve notamment

- les disques durs ;
- les mémoires FLASH, utilisées dans les clés USB, voire dans les disques durs SSD ;
- les disques optiques, comme les CD-ROM, DVD-ROM, et autres CD du genre ;
- les fameuses disquettes, totalement obsolètes de nos jours ;
- mais aussi quelques mémoires très anciennes et rarement utilisées de nos jours, comme les rubans perforés et quelques autres.

Mémoriser un bit

Dans les grandes lignes, les mémoires RAM, ROM, et autres EEPROM actuelles sont toutes composées de **cellules mémoires** capables de retenir un bit. En mettant pleins de ces cellules dans un seul composant, et en mettant quelques circuits électroniques pour gérer le tout, on obtient une mémoire. L'ensemble des cellules mémoires utilisées pour stocker des données s'appelle le **plan mémoire**, et c'est lui qui est l'objet de ce chapitre.

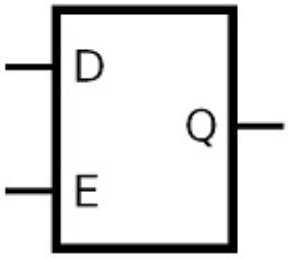
De nos jours, ces cellules mémoires sont fabriquées avec des composants électroniques et il nous faudra impérativement passer par une petite étude de ces composants pour comprendre comment fonctionnent nos mémoires. Dans ce chapitre, nous allons voir ce qu'il y a à l'intérieur d'une mémoire RAM et apprendre à créer nos propres bits de mémoires à partir de composants élémentaires : des transistors. Mais tout d'abord, ce chapitre se limitera aux mémoires de type RAM. En effet, les cellules mémoires de mémoires ROM, PROM, ou EEPROM sont fabriquées d'une façon très différente. Et ne parlons même pas des disques durs qui utilisent le magnétisme pour stocker des données et non des composants électroniques ! Vous verrez aussi que les cellules mémoires des mémoires SRAM et DRAM ne sont pas conçues de la même façon. Bref, commençons !

Mémoire SRAM

Les composants principaux d'une mémoire SRAM sont ce qu'on appelle des bascules. Et oui, il s'agit des fameuses bascules vues au troisième chapitre de ce tutoriel, dans la partie sur les circuits séquentiels. Ce sont de petits composants électroniques capables de mémoriser un bit, et qui peuvent le mettre à jour si besoin. Il existe différents types de bascules qui peuvent être utilisées pour mémoriser des bits (JK, RS à NOR, RS à NAND, RSH, etc), mais on va mettre les choses au point tout de suite : nos mémoires SRAM utilisent toutes une bascule nommé **bascule D**.

Notre bascule est un circuit enfermé dans un boitier, qui contient tout ce qu'il faut pour mémoriser un bit. Mais ce bit ne vient pas de nulle part : notre bascule doit recevoir celui-ci de quelque part. Pour cela, notre bascule possède une entrée sur laquelle on va placer le bit à mémoriser. De même, le bit mémorisé doit pouvoir être lu quelque part et notre bascule va mettre à disposition celui-ci sur une sortie.

Notre bascule commence à se dessiner peu à peu. Voici exactement à quoi elle ressemble.



Le fonctionnement d'une bascule est très simple : quand l'entrée E passe de 1 à 0 (de 0 à 1 sur certaines bascules), le contenu du bit D est recopié sur la sortie Q.

Cette bascule n'est rien d'autre qu'une cellule mémoire de SRAM, qu'on peut lire et écrire à loisir. Si on veut lire le contenu de la bascule, il suffit de lire le bit présent sur la sortie Q. Pour écrire, il faut placer le bit à écrire sur l'entrée D, et faire passer l'entrée E de 0 à 1 : le contenu du bit (la sortie Q) sera alors mit à jour.

Mais à quoi sert l'entrée E ?

Cela permet d'éviter que le bit contenu dans notre cellule de mémoire SRAM soit modifié quand on ne souhaite pas : il faut d'abord autoriser l'écriture (ou la lecture) avec l'entrée E.

Maintenant que l'on sait à quoi ressemble une cellule mémoire vu de l'extérieur, il est temps de passer aux choses sérieuses. Ouvrons cette bascule et regardons ce qu'elle a dans le ventre ! Mais avant de commencer la dissection, autant prévenir tout de suite : il existe plusieurs façons de concevoir une bascule. Aussi, on ne verra que les plus simples.

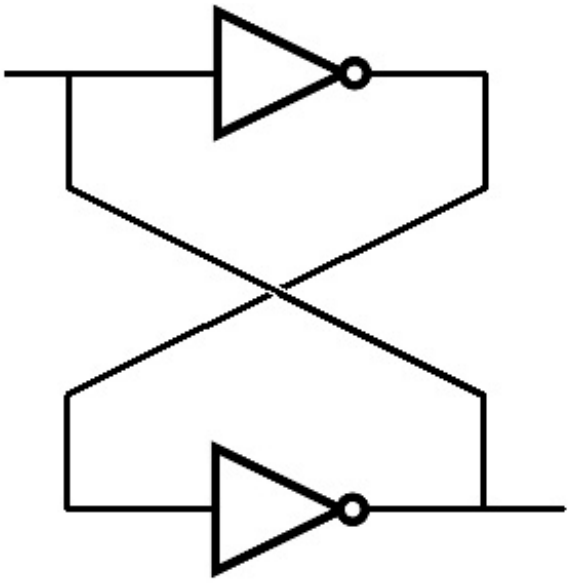
Avec des portes logiques

Nos bascules D peuvent être crées à partir de portes logiques (comme toutes les autres bascules), généralement 2 ou 6. Créer une bascule avec des portes consiste à boucler la sortie d'un circuit sur son entrée, de façon à ce que la sortie rafraîchisse le contenu de l'entrée en permanence et que le tout forme une boucle qui s'auto-entretienne. C'est un des seuls moyen pour créer des mémoires à partir de portes logiques : un circuit qui ne contient pas de boucle, c'est un circuit combinatoire, et ça ne peut

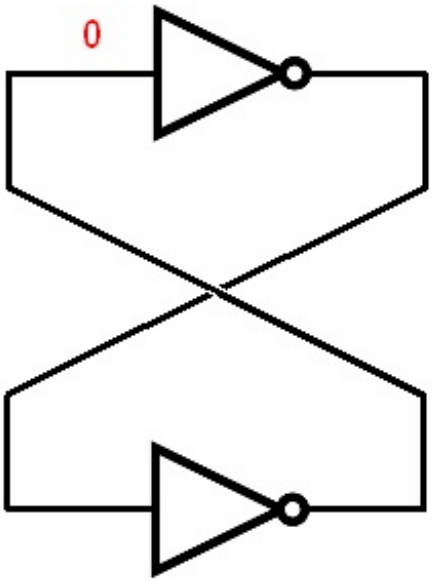
rien mémoriser. Bien sur, cela ne marche pas avec tous les circuits : dans certains cas, cela ne marche pas, ou du moins cela ne suffit pas pour mémoriser des informations. Par exemple, si je relie la sortie d'une porte ***NON*** à son entrée, le montage obtenu ne sera pas capable de mémoriser quoique ce soit.

Et si on essayait avec deux portes ***NON*** ?

Ah, c'est plutôt bien vu ! En effet, en utilisant deux portes ***NON***, et en les reliant comme indiqué sur les schéma juste en dessous, on peut mémoriser un bit.



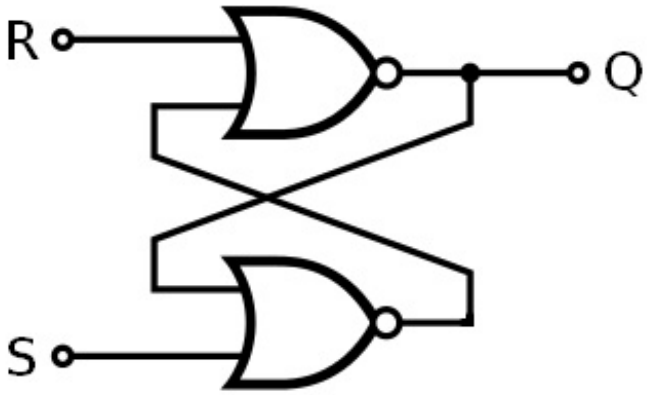
Le premier inverseur va lire le bit mémorisé, et va fournir l'inverse sur sa sortie. Puis, le second inverseur va prendre cet inverse et le ré-inverser encore une fois : on va retrouver le bit de départ sur sa sortie. Cette sortie étant reliée directement sur la sortie Q, on retrouve donc notre bit à mémoriser sur la sortie. L'ensemble sera stable : on peut déconnecter l'entrée du premier inverseur, celle-ci sera alors rafraichie en permanence par l'autre inverseur, avec sa valeur précédente.



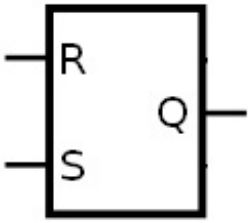
Bascule RS à NOR

Le seul problème, c'est qu'il faut bien mettre à jour l'état de ce bit de temps en temps. Il faut donc ruser. Pour mettre à jour l'état de notre circuit, on va simplement rajouter une entrée à notre circuit qui servira à le mettre à jour, et remplacer notre porte ***NON*** par une porte logique qui se comportera comme un inverseur dans certaines conditions. Le tout est de trouver une porte logique qui inverse le bit venant de l'autre inverseur si l'autre entrée est à zéro (ou à 1, suivant la bascule). Des portes ***NOR*** font très bien

l'affaire.



On obtient alors ce qu'on appelle des **bascules RS**. Celles-ci sont des bascules qui comportent deux entrées R et S, et une sortie Q, sur laquelle on peut lire le bit stocké.



Le principe de ces bascules est assez simple :

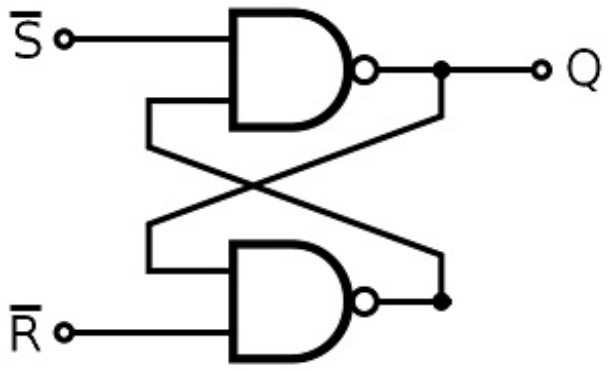
- si on met un 1 sur l'entrée R et un 0 sur l'entrée S, la bascule mémorise un zéro ;
- si on met un 0 sur l'entrée R et un 1 sur l'entrée S, la bascule mémorise un un ;
- si on met un zéro sur les deux entrées, la sortie Q sera égale à la valeur mémorisée juste avant.

Pour vous rappeler de ceci, sachez que les entrées de la bascule ne sont nommées ainsi par hasard : R signifie *Reset* (qui signifie mise à zéro en anglais), et S signifie *Set* (qui veut dire Mise à un en anglais). Petite remarque : si on met un 1 sur les deux entrées, le circuit ne répond plus de rien. On ne sait pas ce qui arrivera sur ses sorties. C'est bête, mais c'est comme ça !

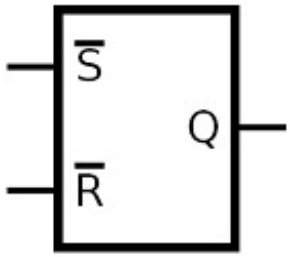
Entrée Reset	Entrée Set	Sortie Q
0	0	Bit mémorisé par la bascule
0	1	1
1	0	0
1	1	Interdit

Bascules RS à NAND

On peut aussi utiliser des portes ***NAND*** pour créer une bascule.



En utilisant des portes ***NAND***, le circuit change un peu. Celles-ci sont des bascules qui comportent deux entrées $\overline{R}$ et $\overline{S}$, et une sortie Q, sur laquelle on peut lire le bit stocké.



Ces bascules fonctionnent différemment de la bascule précédente :

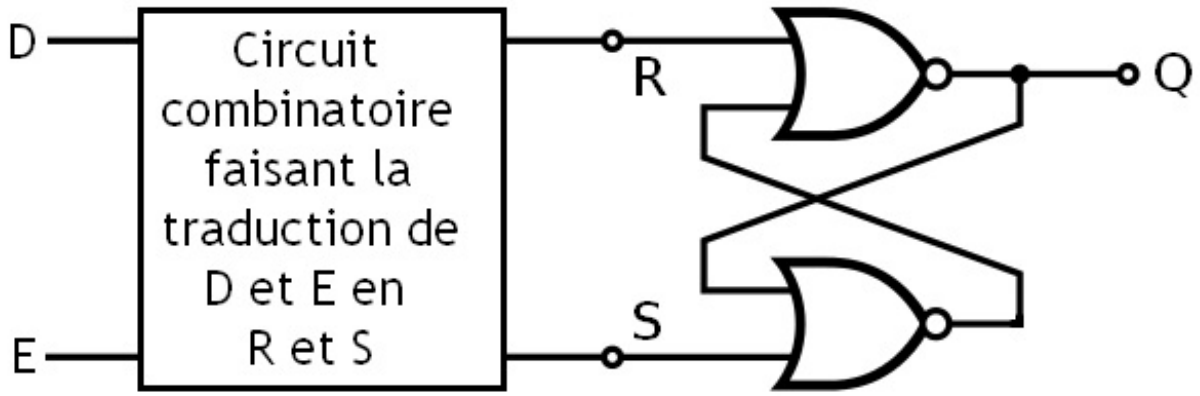
- si on met un 0 sur l'entrée $\overline{R}$ et un 1 sur l'entrée $\overline{S}$, la bascule mémorise un 0 ;
- si on met un 1 sur l'entrée $\overline{R}$ et un 0 sur l'entrée $\overline{S}$, la bascule mémorise un 1 ;
- si on met un 1 sur les deux entrées, la sortie Q sera égale à la valeur mémorisée juste avant.

Entrée Reset	Entrée Set	Sortie Q
0	0	Interdit
0	1	0
1	0	1
1	1	Bit mémorisé par la bascule

Bascule D

C'est à partie de cette bascule RS qu'on va créer une bascule D. Pour créer une bascule D, il suffit simplement de prendre une bascule RS, et de l'améliorer de façon à en faire une bascule D.

Exemple avec une bascule RS utilisant des portes ***NOR***.

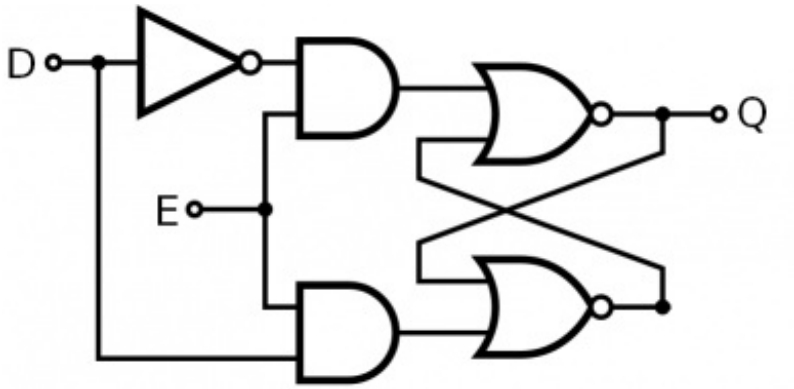


Ce circuit combinatoire est assez simple. Pour le concevoir, il suffit d'écrire sa table de vérité et d'en déduire son équation logique, qu'on traduira en circuit.

Entrée E	Entrée D		Sortie R	Sortie S
0	0		0	0
0	1		0	0
1	0		1	0
1	1		0	1

Ceux qui sont observateurs verront qu'on peut déduire immédiatement l'équation de chaque sortie, R et S. S est égale au résultat d'un simple ***ET*** entre les entrées E et D, tandis que R est égale à $E.\overline{D}$.

On obtient donc ce circuit :

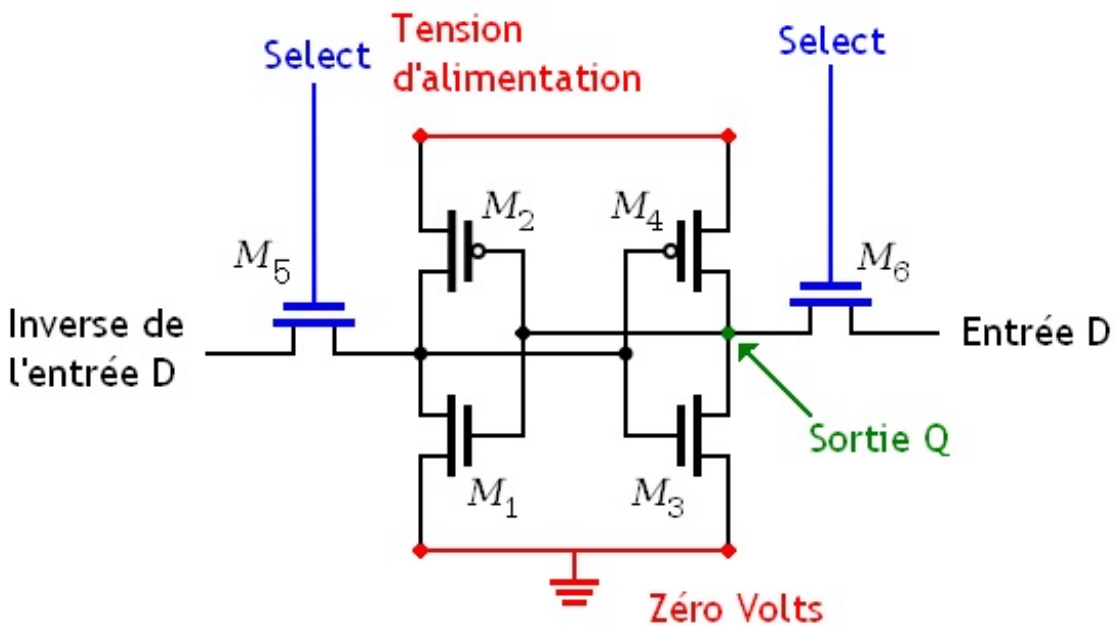


On pourrait faire de même avec une bascule RS utilisant des portes ***NAND***. Il existe de nombreuses autres façons de créer des bascules D avec des portes logiques, mais on ne poursuivra pas plus loin dans cette voie. Il faut dire que les mémoires SRAM de nos ordinateurs ne sont pas vraiment conçues comme ceci.

Avec des transistors

Le schéma vu au-dessus est assez simple à comprendre, mais celui-ci utilise beaucoup de transistors : on tourne autour de 10 à 20 transistors, suivant les transistors et la technologie utilisée. Il y a moyen de faire bien plus simple : certaines mémoires SRAM arrivent à se débrouiller avec seulement 4 ou 2 transistors par bit, ce qui représente une véritable prouesse technique. Cela permet de mettre plus de bits sur la même surface de circuit. Mais les SRAMs utilisées actuellement dans nos processeurs ou dans les mémoires caches utilisent une variante à 6 transistors. Il faut dire que les versions à 1, 2, ou 4 transistors posent quelques problèmes dans des circuits à haute fréquence.

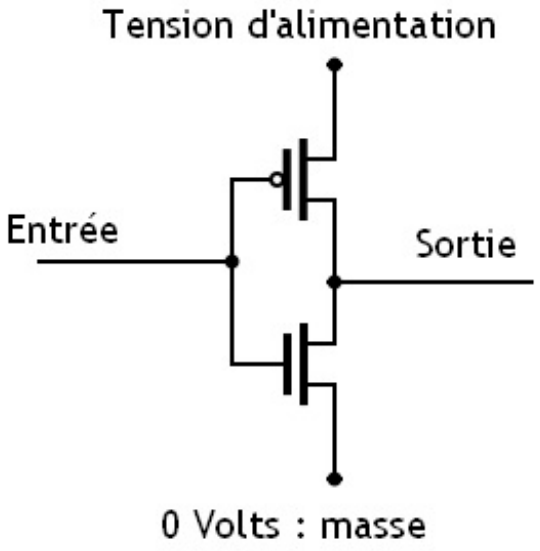
Voici comment sont fabriquées ces bits de SRAM à 6 transistors :



C'est moche, non ?

Bon, c'est pas tout d'avoir balancé un schéma tout moche, il va maintenant falloir que je vous explique comment ça fonctionne. Analysons un peu l'ensemble du circuit, et cherchons à savoir comment tout cela fonctionne.

Tout d'abord, les 4 transistors M1, M2, M3 et M4 sont la cellule mémoire SRAM proprement dite : c'est là qu'est conservé le bit. Si on regarde bien, ce montage est composé de deux sous-circuits composés de deux transistors.



Ce montage a déjà été vu auparavant. Il se contente d'inverser la tension placée sur l'entrée : si cette tension représente un 1, alors la sortie vaut zéro. Et inversement, si l'entrée vaut zéro, la sortie vaut 1. Il s'agit d'un circuit inverseur, aussi appelé une porte ***NON***. Avec ce circuit, la sortie sera connectée soit à la tension d'alimentation et fournira un 1 en sortie, soit à la masse de façon à fournir un zéro en sortie.

Mais ce circuit, tout seul, ne fait qu'inverser le bit passé en entrée. Pour conserver le bit passé en entrée, on utilise deux inverseurs, la sortie du premier étant reliée à l'entrée du second. Et oui, c'est le même principe que pour la création de bascules avec des portes logiques !

Les transistors notés M5 et M6 vont servir d'interrupteur, et relient la cellule mémoire (les 4 transistors du milieu) à l'entrée D. Lors d'une lecture ou d'une écriture, les deux transistors vont s'ouvrir. Cela permet de positionner la sortie des deux inverseurs à la bonne valeur, afin de réaliser une opération d'écriture. Pour une lecture, il suffira de lire le bit voulu sur la sortie Q en fermant

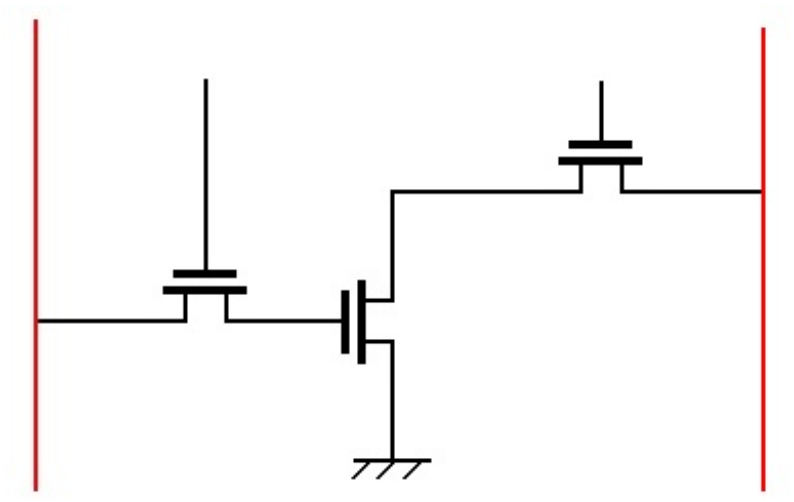
"l'interrupteur" M5. Les mémoires multiports utilisent des transistors supplémentaires, qui ont exactement le même rôle que les transistors M5 et M6, mais qui sont reliés à des entrées et des sorties différentes.

Et là, il faut remarquer un truc : le bit est stocké, mais cela ne fonctionne que tant que nos inverseurs sont alimentés. Si on coupe la tension d'alimentation, la tension à l'intérieur du fil qui distribuait celle-ci dans nos circuit s'annule. En regardant notre montage vu plus haut, on remarque que dans ce cas, la sortie sera reliée soit au zéro volt, soit à une tension d'alimentation nulle : il vaudra toujours zéro et le bit stocké dans ce montage est perdu. Cela explique pourquoi les mémoires SRAM sont des mémoires volatiles.

Mémoire DRAM

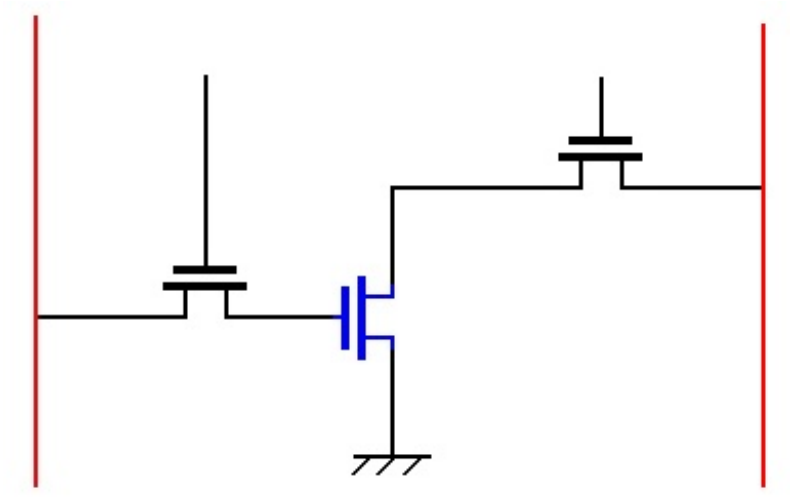
3T-DRAM

Les premières mémoires DRAM fabriquées commercialement utilisaient 3 transistors. Ceux-ci étaitent reliés de cette façon :

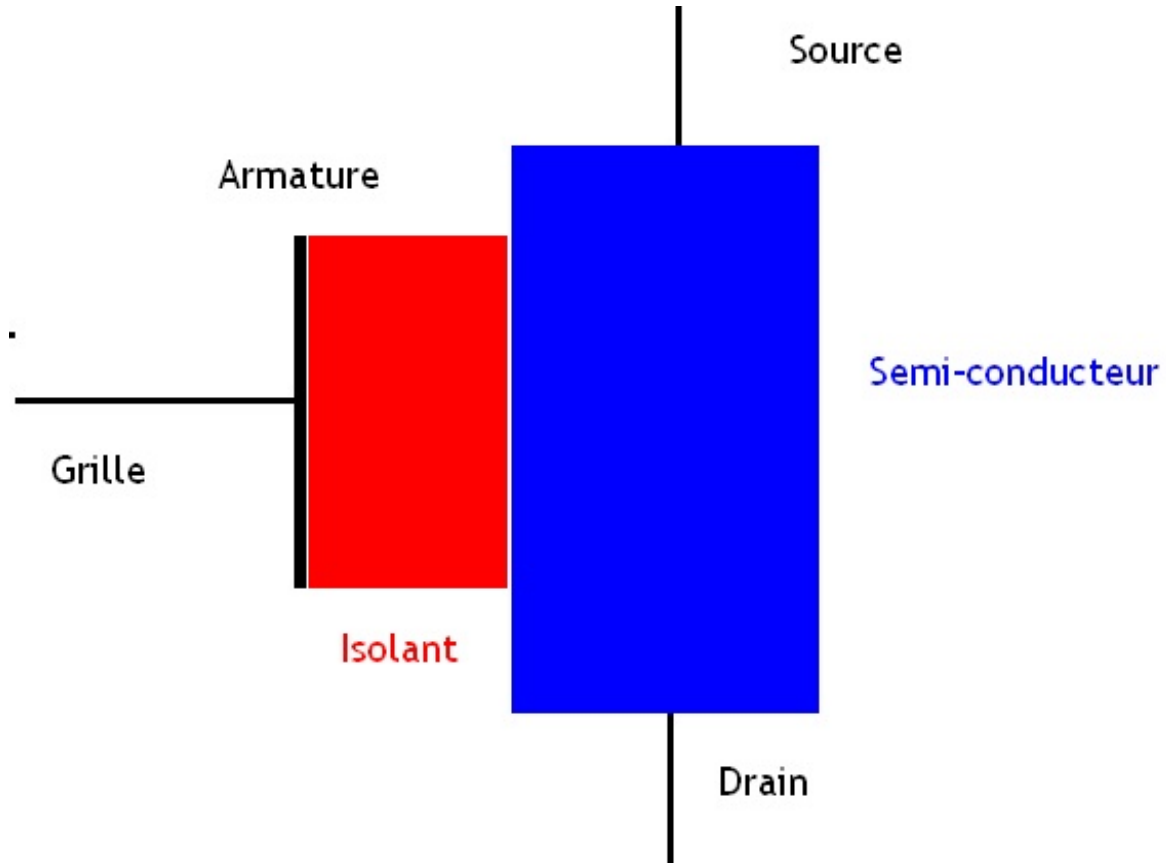


Mémorisation

Le bit est mémorisé dans le transistor du milieu, celui qui est indiqué en bleu sur le schéma :



Cela peut paraitre bizarre : un transistor n'est pas censé pouvoir stocker un bit ! Pour comprendre ce qui se passe, il faut savoir comment fonctionne un transistor CMOS. À l'intérieur du transistor, on trouve simplement une plaque en métal reliée à la grille appelée l'armature, un bout de semi-conducteur entre la source et le drain, et un morceau d'isolant entre les deux.



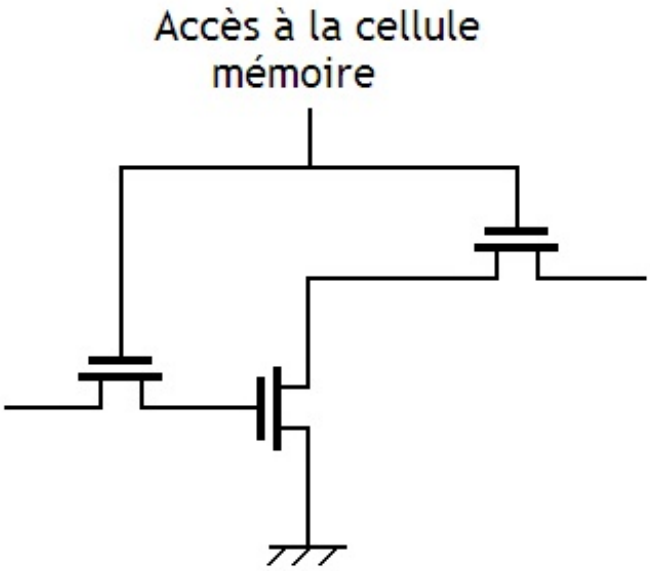
Suivant la tension qu'on envoie sur la grille, l'armature va se remplir d'électrons ou se vider. Et cela nous permet de stocker un bit : il suffit de dire qu'une grille pleine compte pour un 1, et qu'une grille vide compte pour un 0. Bien sûr, cette armature n'est pas parfaite : c'est même une vraie passoire. Celle-ci a tendance à se vide régulièrement et on est obligé de la remettre à jour de temps en temps.

Il faut remarquer qu'avec cette organisation, lire un bit ne détruit pas celui-ci : on peut parfaitement relire plusieurs fois un bit sans que celui-ci ne soit effacé à cause des lectures. C'est une qualité que les DRAM modernes n'ont pas.

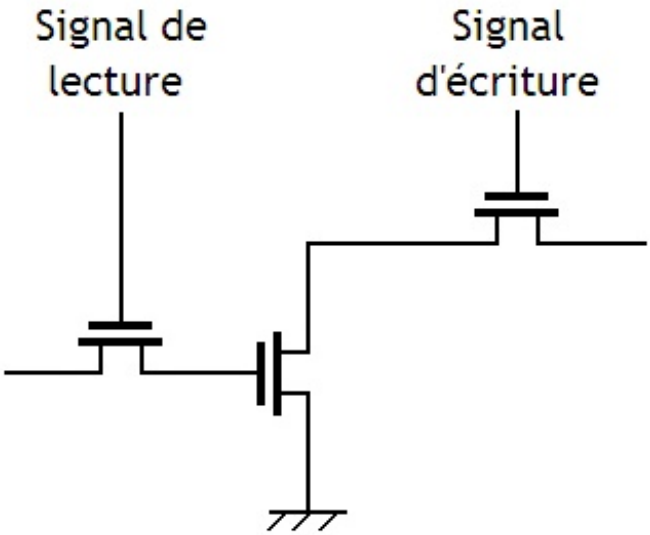
Lectures / écritures

Les deux autres transistors servent à autoriser les lecture et écritures. Ainsi, le transistor de gauche va connecter ou déconnecter le transistor mémorisant notre bit sur la ligne d'écriture. L'autre servira à connecter le transistor stockant notre bit pour effectuer une lecture. Évidemment, il faut bien commander ces deux transistors. Pour cela, il va falloir envoyer un signal qui permettra de demander un accès mémoire, en lecture ou en écriture. Mais cet envoi de signal peut s'effectuer de deux façons.

Soit on utilise un seul signal, qui sert à ouvrir les deux transistors (celui de lecture et celui d'écriture)

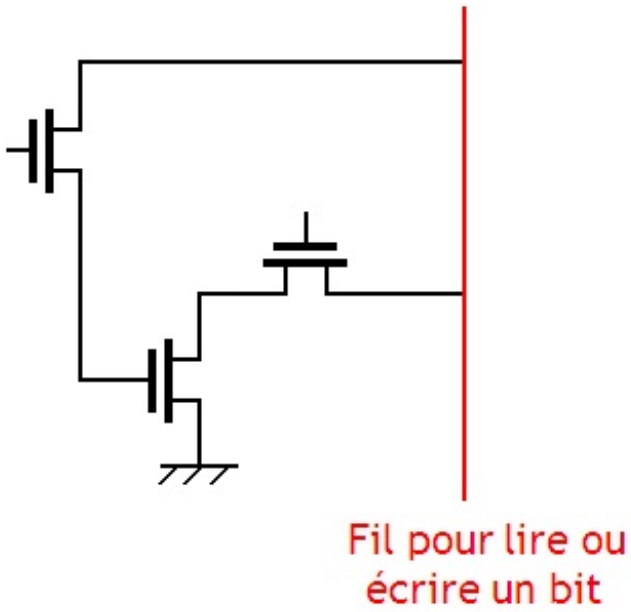


Soit on utilise un signal pour la lecture, et un autre pour la lecture

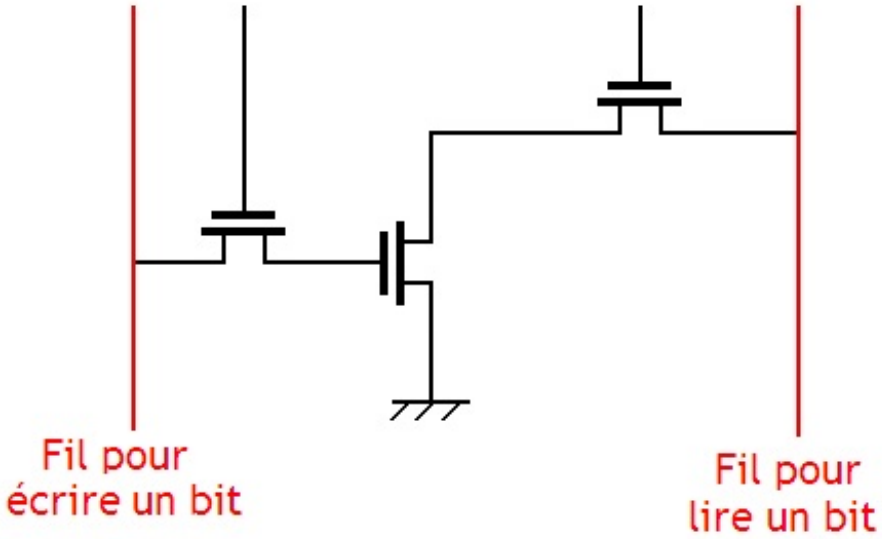


Bit line

Ensuite, dernière différence entre les mémoires DRAM à trois transistor : comment sont reliées les sorties de notre cellules. Sur certaines mémoires DRAM, l'écriture et la lecture d'un bit se font par le même fil : le choix entre lecture et écriture se fait en fonction de ce qu'il y a sur ce fil, et grâce à d'autres portions de la mémoire.



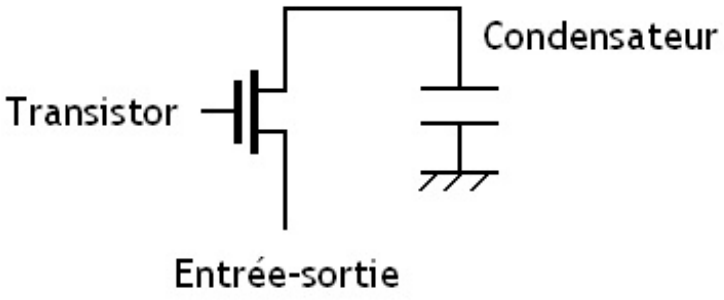
Sur d'autres mémoires, la lecture et l'écriture s'effectuent sur des fils séparés. L'entrée et la sortie de la cellule sont reliées à deux fils séparés.



1T-DRAM

Les DRAM actuelles fonctionnent différemment : elle n'utilisent qu'un seul et unique transistor, et un autre composant électronique nommé un condensateur. Pour simplifier, ce condensateur n'est rien d'autre qu'un gros réservoir à électrons : on peut le remplir d'électrons ou le vider en mettant une tension sur ses entrées. C'est ce condensateur qui va stocker notre bit : le condensateur stocke un 1 s'il est rempli, et stocke un 0 s'il est vide. Rien de plus simple.

A coté, on ajoute un transistor qui relie ce condensateur au reste du circuit. Ce transistor sert d'interrupteur : c'est lui qui va autoriser l'écriture ou la lecture dans notre condensateur. Tant que notre transistor se comporte comme un interrupteur ouvert, le courant ne passe pas à travers, et le condensateur est isolé du reste du circuit. : pas d'écriture ou de lecture possible. Si on l'ouvre, on pourra alors lire ou écrire dedans.



On utilise seulement un transistor et un condensateur. Une DRAM peut stocker plus de bits pour la même surface qu'une SRAM grâce à cela : à votre avis, entre un transistor couplé à un condensateur et 6 transistors, qui prend le moins de place ?

Lecture et écriture d'un bit

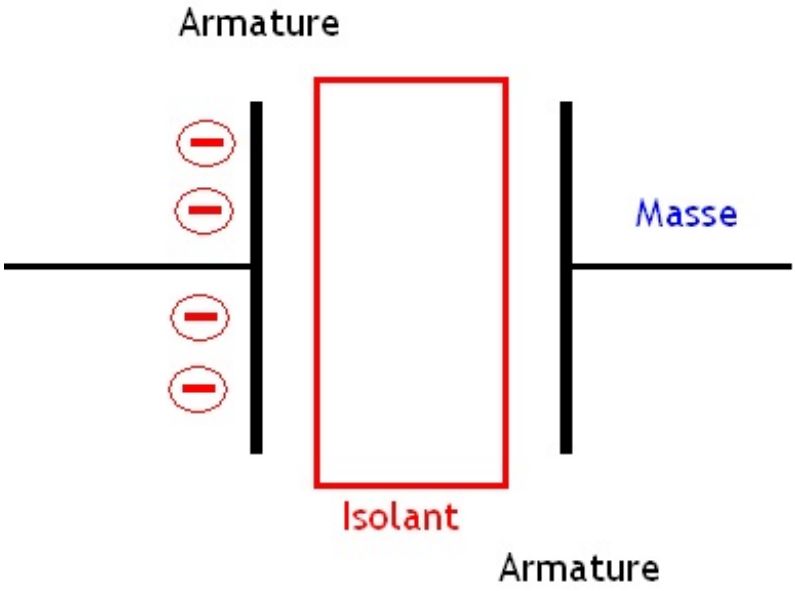
Seul problème : quand on veut lire ou écrire dans notre cellule mémoire, le condensateur va être connecté sur le bus de donnée. Et quand le condensateur est connecté à un bout de fil, il se vide entièrement ! On perd son contenu : il faut donc le récrire après chaque lecture.

Pire : le condensateur se vide sur le bus, mais cela ne suffit pas à créer une tension de plus de quelques millivolts dans celui-ci. Pas de quoi envoyer un 1 sur le bus ! Mais il y a une solution : amplifier la tension de quelques millivolts induite par la vidange du condensateur sur le bus. Pour cela, il faut donc placer un dispositif capable d'amplifier cette tension.

Une vraie passoire !

Il faut préciser une chose sur notre condensateur : celui-ci est plus proche d'une passoire que d'un réservoir à électrons. Un condensateur n'est pas vraiment un réservoir parfait, qui stockerait des électrons indéfiniment : il possède toujours quelques défauts et des imperfections qui font que celui-ci se vide tout seul a bout d'un moment.

Pour expliquer pourquoi, il faut savoir comment est fabriqué en condensateur. Celui-ci n'est rien d'autre qu'un ensemble de morceaux de conducteur électrique (du métal) séparés par un isolant. Chacun de ces morceaux étant appelé une armature. C'est sur une de ces armatures, que vont s'accumuler les électrons, l'autre armature étant reliée à un fil contenant une tension de zéro volts : la masse.



Logiquement, l'isolant empêche les électrons de passer d'une armature à l'autre : ces électrons n'ont nulle part où aller et sont censés rester sur l'armature tant que le transistor servant d'interrupteur ne décide de relier le condensateur au reste du circuit. Mais dans la réalité, l'isolant qui sépare les deux armatures n'est pas totalement étanche, et des électrons passent de l'armature qui le stocke à l'autre, reliée à la masse, et quittent donc le condensateur.

En clair, le bit contenu dans la cellule de mémoire DRAM s'efface, et c'est pour cela qu'on doit le récrire régulièrement. Vous

comprenez maintenant pourquoi on doit rafraichir une mémoire DRAM, et aussi pourquoi celle-ci est volatile.

Correction d'erreurs

Une mémoire n'est pas un dispositif parfait : il est possible que certaines opérations de lecture ou d'écriture ne se passent pas correctement et qu'on lise ou écrive des données corrompues. Un bit d'une mémoire peut parfaitement être modifié, suite à l'action d'un rayonnement cosmique ou d'une perturbation électromagnétique de passage. Après tout, ce n'est pas une chose si rare : on est sans cesse entouré par des rayonnements divers, aussi bien naturels qu'artificiels, qui peuvent interférer avec le fonctionnement des appareils électroniques qui nous entourent et les mémoires ne font pas exception !

Pour donner un exemple, on peut citer l'incident de Schaerbeek. Le 18 mai 2003, dans la petite ville belge de Schaerbeek, une défaillance temporaire d'une mémoire faussa les résultats d'une élection. Cette ville utilisait une machine à voter élèctronique, qui contenait don forcément une mémoire. Et on constata un écart de 4096 voix en faveur d'un candidat entre le dépouillement traditionnel et le dépouillement élèctronique. Mais ce n'était pas une fraude : le coupable était un rayon cosmique, qui avait modifié l'état d'un bit de la mémoire de la machine à voter.

Cet incident n'était pas trop grave : après tout, il a pu corriger l'erreur. Mais imaginez la même défaillance dans un système de pilotage en haute altitude...

Correction et détection d'erreurs

Heureusement, certaines mémoires sont capables de limiter les effets de ces erreurs en les détectant, voire en les corrigeant. Oui, vous avez bien lu : seules certaines mémoires spécialement conçues pour en sont capables. Et c'est tout à fait normal : dans la majorité des cas, on se moque qu'un bit de notre mémoire aie été modifié, cela ayant peu de conséquences. Seules certaines applications critiques qui ne tolèrent pas la moindre erreur, comme les ordinateurs implantés sur des serveurs ou des satellites, sont concernés par ce genre de problèmes.

Dans les autres cas, utiliser une mémoire capable de corriger ou de détecter des erreurs est inutile. Sans compter que les mémoires capables de corriger des erreurs sont plus chères et parfois plus lentes que les mémoires ordinaires : il faut bien rajouter des circuits capables de détecter ou de corriger ces erreurs, et cela a un cout pas vraiment négligeable en terme d'argent ou de performances !

Quoi qu'il en soit, certaines mémoires utilisent des techniques plus ou moins évoluées pour corriger les erreurs et autres corruptions de la mémoire. Ces techniques nécessitent toutes l'ajout de bits supplémentaires pour pouvoir fonctionner : chaque case mémoire contient non seulement les bits qui servent à coder une donnée, mais aussi des bits cachés qui servent uniquement à détecter ou corriger des erreurs. Ces bits ne sont jamais reliés au bus de donnée, et sont accessibles et modifiables uniquement par des circuits internes à la mémoire qui sont spécialement dédiés au contrôle et à la correction des erreurs.

Bit de parité ou d'imparité

La première technique de correction d'erreur n'est rien d'autre que le fameux bit de parité, vu au chapitre 3. Pour rappel, ce bit de parité permet de détecter des erreurs qui modifient un nombre impair de bits. Si un, trois, cinq, ou un nombre impair de bits voient leur valeur s'inverser (un 1 devient un 0, ou un 0 qui devient un 1), la technique du bit de parité (ou d'imparité) permettra de détecter cette erreur. Par contre, il sera impossible de la corriger.

Le principe caché derrière un bit de parité est simple : il suffit d'ajouter un bit supplémentaire aux bits à stocker. Ce bit, **le bit de parité** vaudra zéro si le nombre de bits à 1 dans le nombre à stocker (bit de parité exclu) est pair, et vaudra 1 si ce nombre est impair. Le but d'un bit de parité est de faire en sorte que le nombre de bits à 1 dans le nombre à stocker, bit de parité inclut, soit toujours un nombre pair : si cette somme est paire, on rajoute zéro, et si elle est impaire on rajoute un un pour obtenir un nombre de bit à 1 pair.

De même, il existe un bit d'imparité, conçu de façon à ce que le nombre de bits à 1 dans le nombre, bit d'imparité inclut, soit un nombre impair. Sa valeur est l'exact inverse de celle d'un bit de parité obtenu pour le même nombre.

Exemple

Prenons le nombre **00000101**. Celui-ci contient 6 bits à 0 et 2 bits à 1. La somme de tous ces bits vaut donc 2. Le bit de parité vaudra donc zéro.
En plaçant le bit de parité au début du nombre, on obtient : **000000101**.

Autre exemple : le nombre **11100101**. Celui-ci contient 3 bits à 0 et 5 bits à 1. On trouve 5 bits à 1 dans ce nombre, ce qui donne un nombre impair. Le bit de parité vaudra donc un.
Le total sera donc : **11110101**.

Détection d'erreur

Détecter une erreur est simple : on compte le nombre de bits à 1 dans le nombre à stocker, bit de parité inclut, et on regarde s'il est pair. S'il est impair, on sait qu'au moins un bit à été modifié. En modifiant un bit, la parité du nombre total de bits à 1 changera : le nombre de bits à 1 sera amputé (si un 1 devient un 0) ou augmenté de 1 (cas inverse) et deviendra un nombre impair. Et ce qui est valable pour un bit l'est aussi pour 3, 5, 7, et pour tout nombre impair de bits modifiés. Par contre, si un nombre pair de bit est modifié, la parité du total ne changera pas et restera compatible avec la valeur du bit de parité : on ne pourra pas détecter l'erreur.

Mémoires ECC

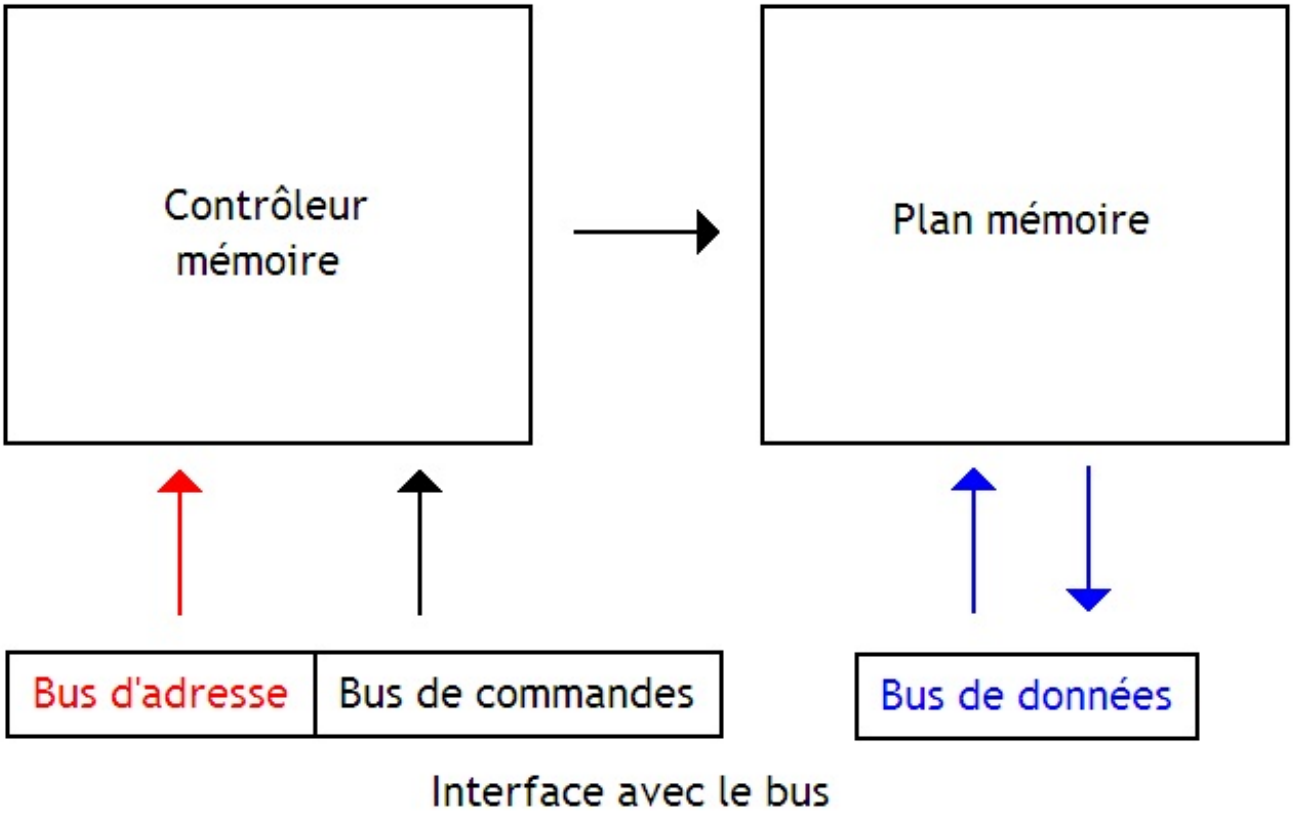
Mais savoir qu'un bit a été modifié sans pouvoir corriger l'erreur est quelque peu frustrant. Sans compter les erreurs qui modifient un nombre pair de bits, qui ne sont pas détectées. Pour résoudre ces défauts, inhérents aux mémoires utilisant un bit de parité, on a inventé d'autres types de mémoires : les **mémoires ECC**. Ces mémoires utilisent des méthodes de détection d'erreur plus sophistiquées, qui utilisent au moins deux bits supplémentaires par case mémoire. On les retrouve le plus souvent dans des serveurs ou dans des ordinateurs qui doivent fonctionner dans des environnements hostiles sans tomber un panne : la mémoire d'un calculateur implanté dans un satellite ou une navette spatiale.

Le plus souvent, la technique de correction d'erreur utilisé est un **code de hamming**, couplé avec un bit de parité. Cela permet de détecter les erreurs qui modifient 2 bit ou un nombre impair de bits, et de corriger les erreurs qui ont modifiées 1 bit.

Contrôleur et plan mémoire

Une fois qu'on a réussi à créer des cellules mémoires, celles-ci ne nous servent à rien si l'on ne peut pas les sélectionner. Heureusement, les mémoires actuelles sont adressables, et on peut préciser quelle case mémoire lire ou écrire en précisant son adresse. Cette gestion de l'adresse mémoire ne se fait pas toute seule : vous vous doutez bien qu'on a forcément besoin de circuits supplémentaires pour gérer l'adressage et la communication avec le bus. Ce rôle est assuré par un circuit spécialisé qu'on appelle le contrôleur mémoire.
Notre mémoire est ainsi composée :

- d'un tas de cellules mémoires capables de retenir 1 bit, regroupées dans un **plan mémoire** ;
- d'un circuit qui gère le plan mémoire, nommé le **contrôleur mémoire** : il s'occupe de l'adressage, du rafraichissement mémoire (pour les DRAM) et de bien d'autres choses ;
- et des **connexions avec le bus**.



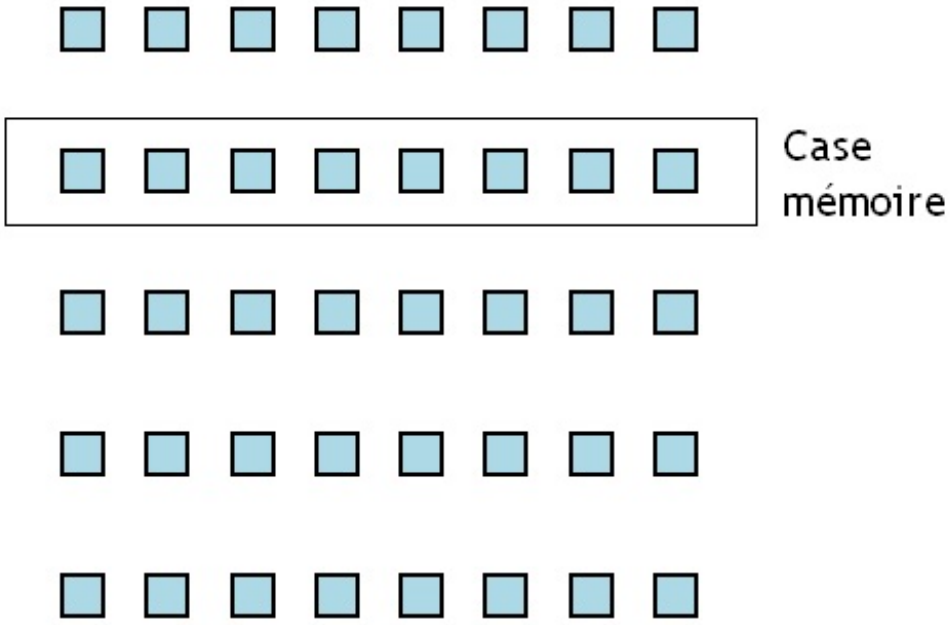
Dans ce chapitre, on va voir comment l'ensemble est organisé, et voir dans les grandes lignes comment fonctionne un contrôleur mémoire. Mais avant tout, je tiens à donner une petite précision : je ne parlerais pas du fonctionnement des mémoires multiports dans ce chapitre, et me contenterais d'évoquer les mémoires connectées à un seul bus.

Mémoires à adressage linéaire

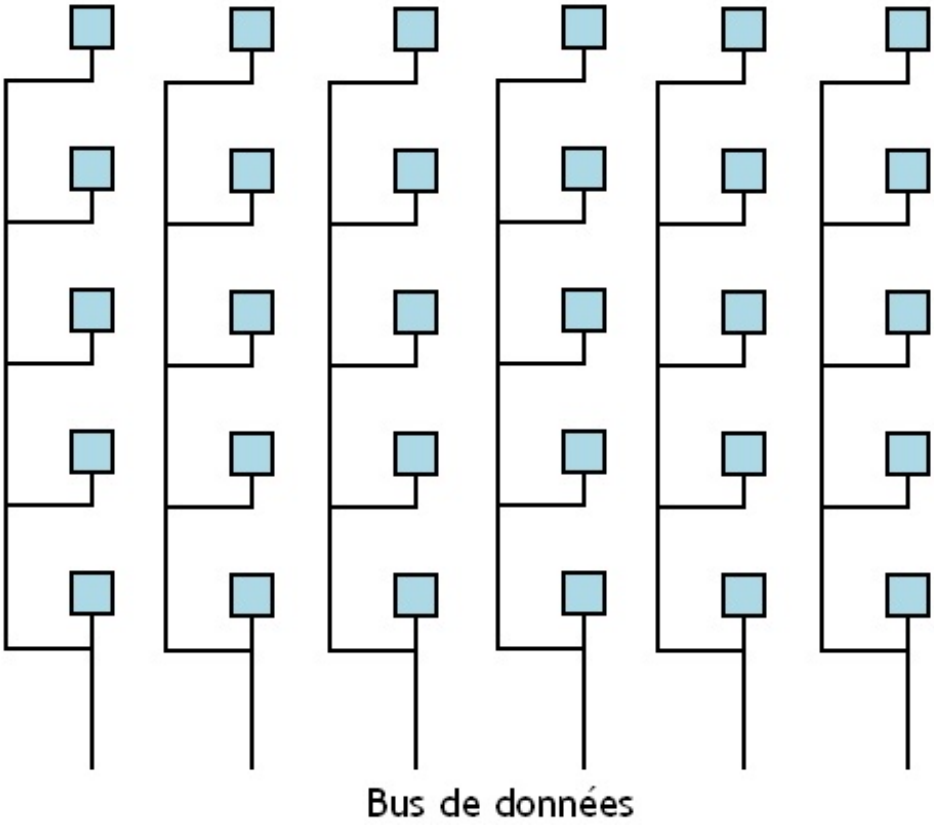
Pour commencer, il faut savoir que tous les plans mémoires ne se ressemblent pas. Il y a beaucoup de façons plus ou moins efficaces d'organiser nos cellules mémoires. Nous allons commencer par parler des plans mémoires les plus simples : ceux utilisées dans ce qu'on appelle les **mémoires à adressage linéaire**. Pour vous donner un exemple, les registres du processeur sont le meilleur exemple possible de mémoire à adressage linéaire.

Plan mémoire linéaire

Sur de telles mémoires, le plan mémoire est organisé sous la forme d'un tableau rectangulaire de cellules mémoires. Les cellules mémoires sont alignées les unes à coté des autres, aussi bien horizontalement que verticalement. Toutes les cellules mémoires placées sur une ligne appartiennent à une même case mémoire.



Chaque cellule mémoire d'une case mémoire est connectée sur un fil qui lui permettra de communiquer avec le bus de donnée. Chacun de ces fils s'appelle la **Bit Line**. Avec cette organisation, la cellule mémoire stockant le i-éme bit du contenu d'une case mémoire (le bit de poids i) est reliée au i-éme fil du bus.



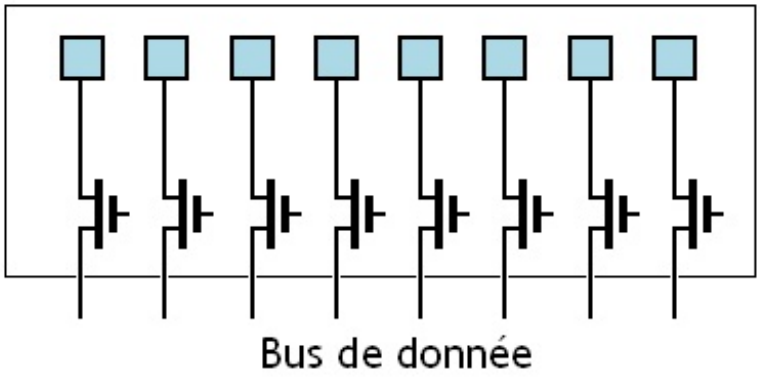
Connectons le tout au bus

Reste un premier problème : comment sélectionner la bonne case mémoire à lire ou écrire ?

Vu qu'une case mémoire est stockée sur une ligne, il suffit de sélectionner la bonne ligne dans le plan mémoire . Pour pouvoir

sélectionner une ligne, une solution simple est utilisée : on déconnecte du bus les cases mémoires/lignes dans lesquelles on ne veut pas écrire ou lire, et on connecte sur le bus uniquement la case mémoire à manipuler. Cette connexion/déconnexion est réalisée par un vulgaire interrupteur qu'on peut commander électroniquement (pour lui dire de s'ouvrir ou de se fermer) : notre transistor fait son retour !

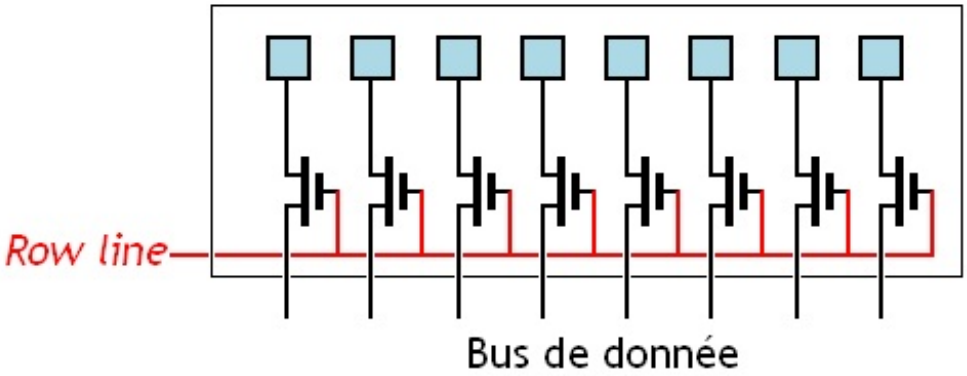
Exemple en utilisant un transistor en guise d'interrupteur.



Pour autoriser une lecture ou une écriture dans une cellule mémoire, il suffira de fermer ce transistor en envoyant un 1 sur la grille de celui-ci. Par contre, notre cellule mémoire sera déconnectée du bus si la grille du transistor est à zéro : le transistor se comportera comme un interrupteur ouvert.

Vous vous souvenez que dans les cellules mémoires de DRAM (et certaines SRAM), il y a au moins un petit circuit (souvent un transistor) qui permet d'autoriser les lectures ou écriture ? Et bien il suffit d'intercaler ce transistor entre le bus et la mémoire. Dans ce genre de cas, le transistor est intégré à la cellule mémoire. Mais ça n'est pas toujours le cas.

Vu que tous les bits d'une case mémoire sont rassemblés sur une ligne, tous les transistors reliés aux cellules mémoires de cette ligne devront "s'ouvrir" ou se "fermer" en même temps : on relie donc leur grille au même fil, sur lequel on enverra un 1 ou un zéro selon qu'on veuille ouvrir ou fermer ces transistors.



Comme vous le voyez, ce fil s'appelle ***Row Line*** sur le schéma. Pour sélectionner notre case mémoire parmi toutes les autres, il suffira de positionner son entrée *Row Line* à 1, et placer les entrées *Row Line* de toutes les autres cases mémoires à zéro. Le rôle du contrôleur mémoire est donc de **déduire quelle entrée *Row Line* mettre à un à partir de l'adresse** envoyée sur le bus d'adresse.

Décodeurs

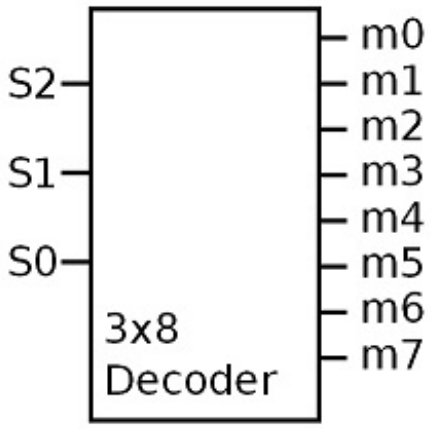
Pour sélectionner la bonne ligne, notre contrôleur mémoire doit répondre à plusieurs exigences :

- Il doit partir d'une adresse codée sur n bits, et en déduire quelle case mémoire sélectionner : ce contrôleur a donc n entrées ;
- notre adresse de n bits peut adresser 2^n bytes : notre contrôleur mémoire doit donc posséder 2^n sorties ;
- chacune de ces sorties sera reliée à une entrée *row line* et permettra de connecter ou déconnecter une case mémoire du

bus ;
- on ne doit sélectionner qu'une seule case mémoire à la fois : une seule sortie devra être placée à 1, et toutes les autres à zéro ;
- et enfin, deux adresses différentes devront sélectionner des cases mémoires différentes : la sortie de notre contrôleur qui sera mise à 1 sera différente pour deux adresses différentes placées sur son entrée.

Il existe un composant électronique qui répond à ce cahier des charges : le **décodeur**. C'est le composant parfait pour positionner notre bit *Row Line*.

Exemple d'un décodeur à 3 entrée et 8 sorties.



On peut remarquer que toutes les sorties d'un décodeur sont numérotées : sur un décodeur possédant N entrées, nos sorties sont numérotées de 0 à $2^N - 1$. Le fonctionnement de notre décodeur est très simple : il prend sur son entrée un nombre entier **x** codé en binaire, positionne à 1 la sortie numéroté **x** et positionne à zéro toutes les autres sorties.

Avec tables de vérités

Ce décodeur est, comme tous les autres circuits électroniques, conçu avec des portes logiques. Dans sa version la plus naïve, on peut créer un décodeur en utilisant les techniques vues au chapitre 3 : on établit une table de vérité, qu'on transforme en équations logiques, et on traduit le tout en circuit.

Pour donner un exemple, nous allons montrer l'exemple d'un décodeur 2 vers 4. Commençons d'abord par écrire sa table de vérité.

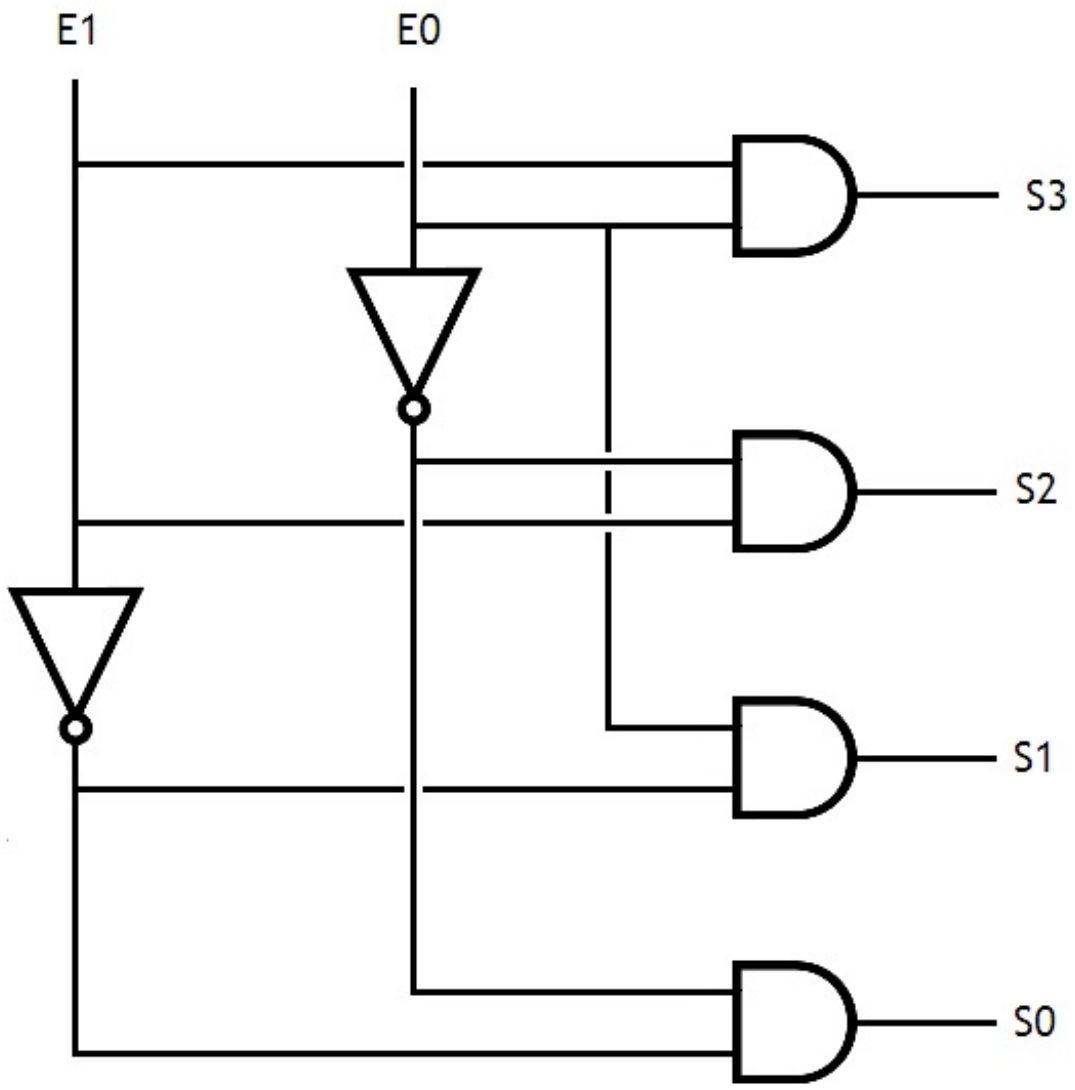
Entrée A1	Entrée A0	-	Sortie S3	Sortie S2	Sortie S1	Sortie S0
0	0	-	0	0	0	1
0	1	-	0	0	1	0
1	0	-	0	1	0	0
1	1	-	1	0	0	0

Comme vous le voyez, on se retrouve avec nos sorties à 1 sur une diagonale. Et cela a une conséquence : cela signifie qu'une fois nos équations logiques écrites, il sera impossible de les simplifier ! On se doute bien qu'à cause de cela, notre décodeur va utiliser beaucoup de portes logiques.

Quoiqu'il en soit, on obtient alors les équations logiques suivantes :

- $S3 = \overline{E1}.\overline{E0}$;
- $S2 = \overline{E1}.\overline{E0}$;
- $S1 = \overline{E1}.\overline{E0}$;
- $S0 = \overline{E1}.\overline{E0}$;

Une fois traduite en circuit, on obtient alors le circuit suivant :

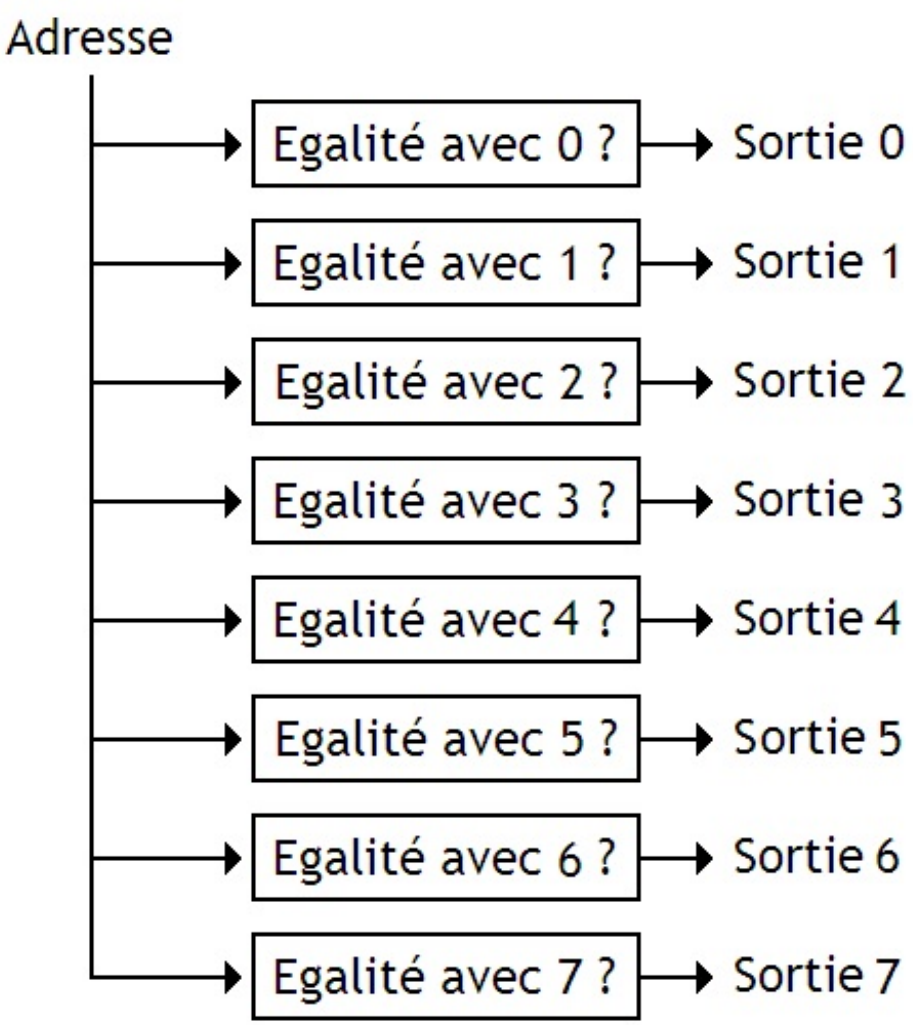


Décodeurs complets

Utiliser une table de vérité pour créer un décodeur est impossible si celui-ci utilise trop d'entrées. Néanmoins, il est possible de se passer de celle-ci en rusant un peu.

En réfléchissant bien, on sait qu'on peut déduire la sortie assez facilement en fonction de l'entrée. Si l'entrée vaut 0, la sortie mise à 1 sera la sortie 0. Si l'adresse vaut 1, ce sera la sortie 1. Et on peut continuer ainsi de suite. En clair : si l'adresse vaut N, la sortie mise à 1 sera la sortie N. Bref, déduire quand mettre à 1 la sortie N est facile : il suffit de comparer l'adresse avec N. Si l'adresse vaut N, on envoie un 1 sur la sortie, et on envoie un zéro sinon. Pour cela, j'ai donc besoin d'un comparateur spécial pour chaque sortie, et le tour est joué.

Exemple d'un décodeur à 8 sorties.



En faisant cela, on se retrouve avec un circuit très similaire à ce qu'on aurait obtenu en utilisant une table de vérité. A quelques détails prêts, bien sûr.

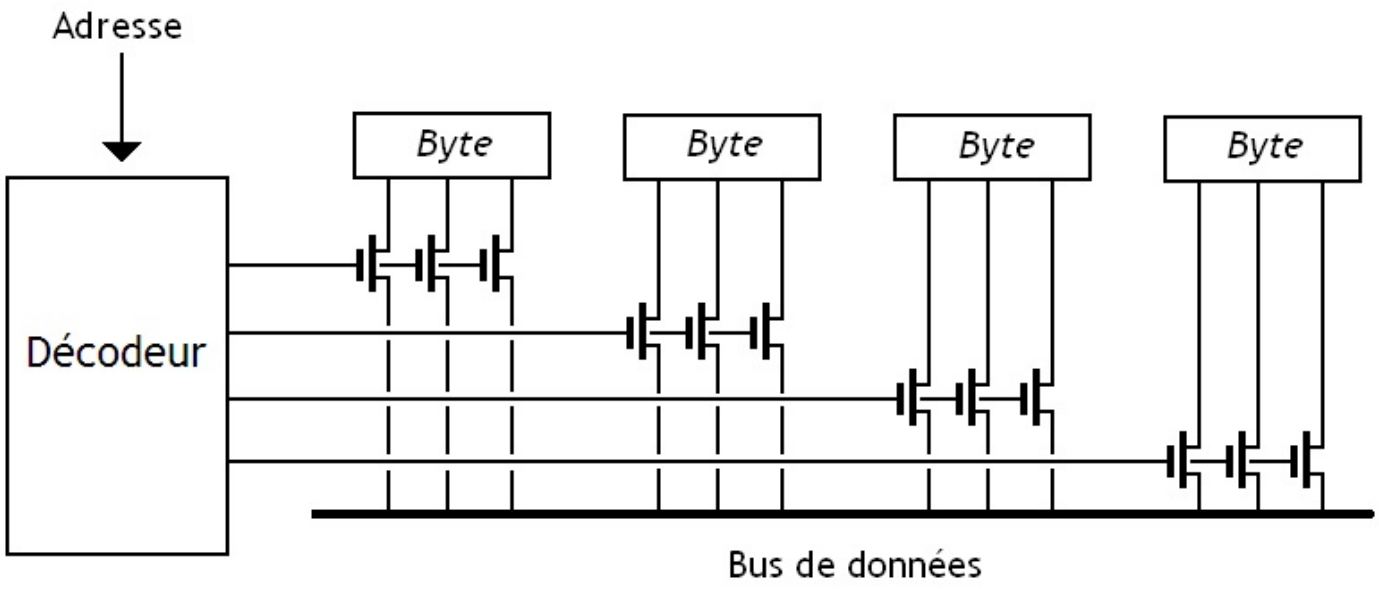
Pré-décodage

Si on crée un décodeur à partir des techniques vues au-dessus, ce décodeur utilisera une grande quantité de portes logiques. Plus précisément, la quantité de portes logiques utilisée augmentera exponentiellement avec le nombre d'entrées : cela devient rapidement inutilisable. Pour éviter cela, il existe d'autres types de décodeurs, qui utilisent moins de portes logiques. Ces derniers sont toutefois plus lents que leurs congénères crées à partir d'une table de vérité. Ces décodeurs moins gourmands en circuits ont un câblage assez complexe, aussi on vous l'épargnera dans ce qui va suivre.

Circuit complet

Pour adresser une mémoire à adressage linéaire, il suffit de relier chaque fil *Row Line* sur une sortie de ce décodeur. Plus précisément, le bit *Row Line* de la case mémoire d'adresse N doit être connectée à la sortie numéro N du décodeur : ainsi, si on envoie l'adresse N sur l'entrée de ce décodeur, la sortie N sera sélectionnée, ce qui sélectionnera la case mémoire appropriée. L'adresse a juste à être placée directement sur l'entrée de notre décodeur, qui se chargera de sélectionner le bon fil *Row Line* qui correspond à notre adresse.

Exemple d'une mémoire de 4 cases mémoires avec une adresse de deux bits.



Notre contrôleur mémoire se résumera donc à un simple décodeur, avec quelques circuits pour gérer le sens de transfert (lecture ou écriture), et la détection/correction d'erreur. Ce genre d'organisation s'appelle l'**adressage linéaire**.

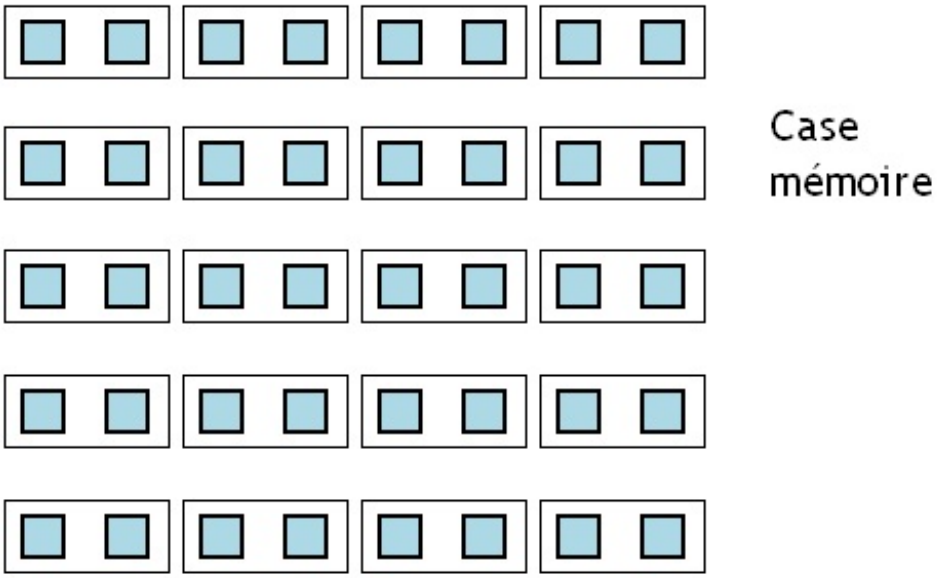
Mémoires à adressage par coicidence

Sur des mémoires ayant une grande capacité, utiliser un seul gros décodeur va poser quelques problèmes. En effet, plus notre décodeur doit adresser de cases mémoires, plus celui-ci va contenir de portes logiques. Et ce nombre risque d'augmenter un peu trop. Et c'est sans compter que ce gros décodeur risque d'être lent, pour des tas de raisons techniques. Or, la taille d'un décodeur dépend fortement du nombre de *Row Line* qu'il doit commander. Pour garder un décodeur petit, on doit absolument diminuer le nombre de lignes tout en gardant la taille d'un *byte* intact.

De plus, diminuer le nombre de lignes a un gros avantage. Cela permet de diminuer la longueur des *Bit Lines*. Il faut dire que plus un fil est long, plus la tension et le courant vont mettre du temps pour passer d'un bout à l'autre du fil. Ce temps de propagation des tensions et courants dans le fil va ralentir assez fortement la vitesse de notre mémoire. Plus on connectera de *Bytes* sur notre *Bit Line*, pire ce sera.

Principe

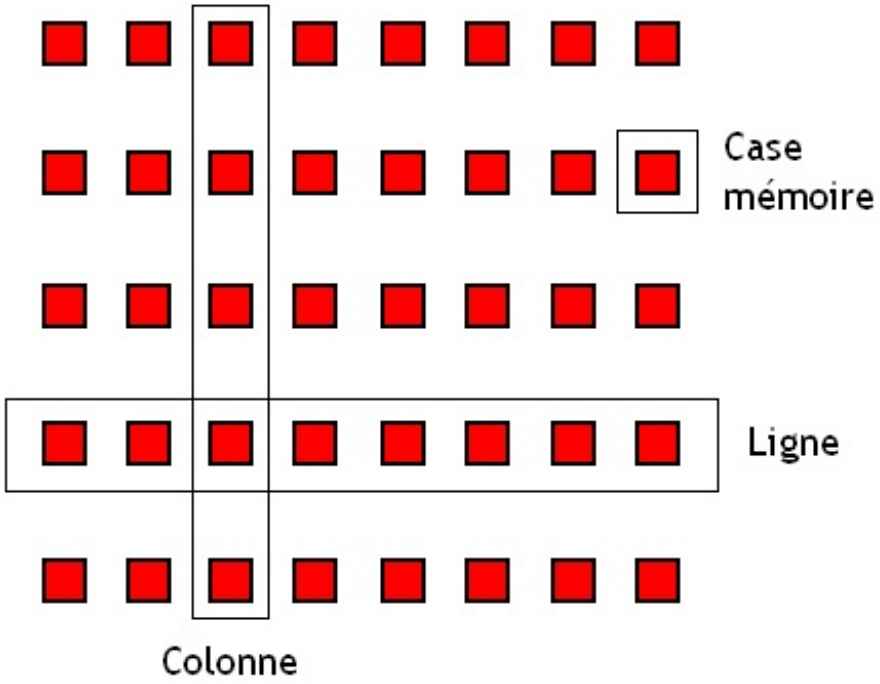
Vous l'avez compris, diminuer le nombre de lignes dans notre mémoire n'est que de la légitime défense. Et il n'y a qu'une seule solution à cela : regrouper plusieurs cases mémoires sur une seule ligne.



Ainsi, il suffira de sélectionner la ligne voulue, et sélectionner la case mémoire à l'intérieur de la ligne. Sélectionner une ligne est facile : on utilise un décodeur. Mais la sélection de la colonne est quelque chose de nettement plus intéressant : la méthode utilisée pour sélectionner la colonne dépend de la mémoire utilisée. Commençons par aborder la première méthode : celle utilisée pour les **mémoires à adressage par coïncidence**.

Adressage par coïncidence

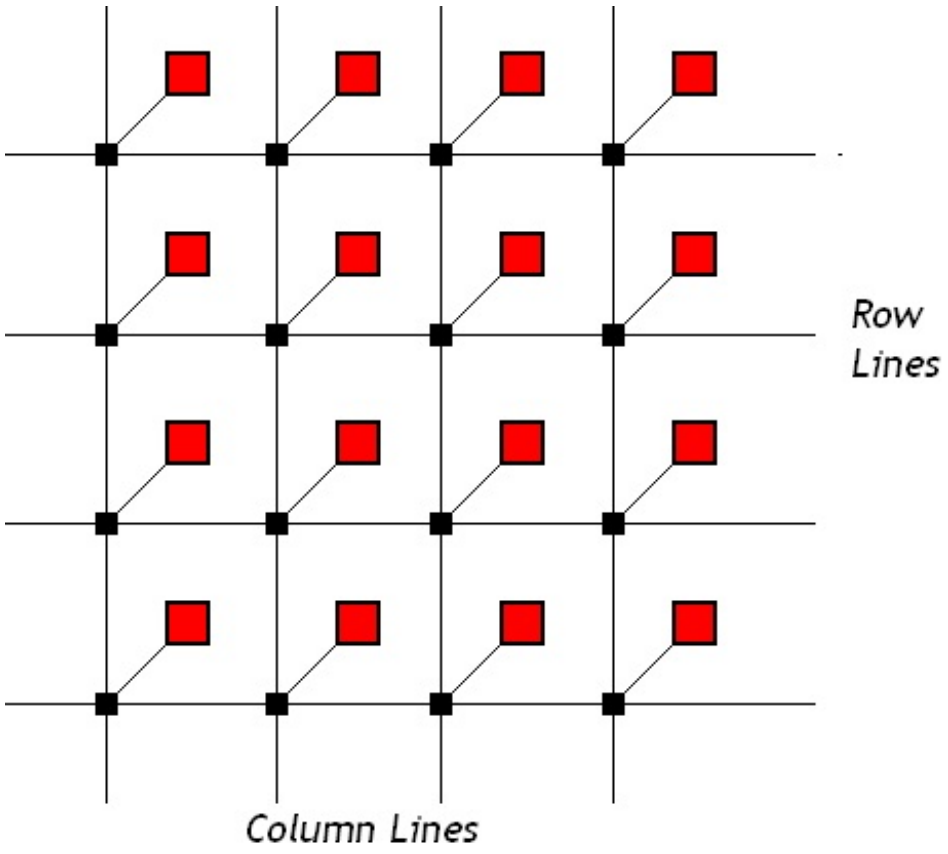
Sur ces mémoires, les cases mémoires sont organisées en lignes et en colonnes.



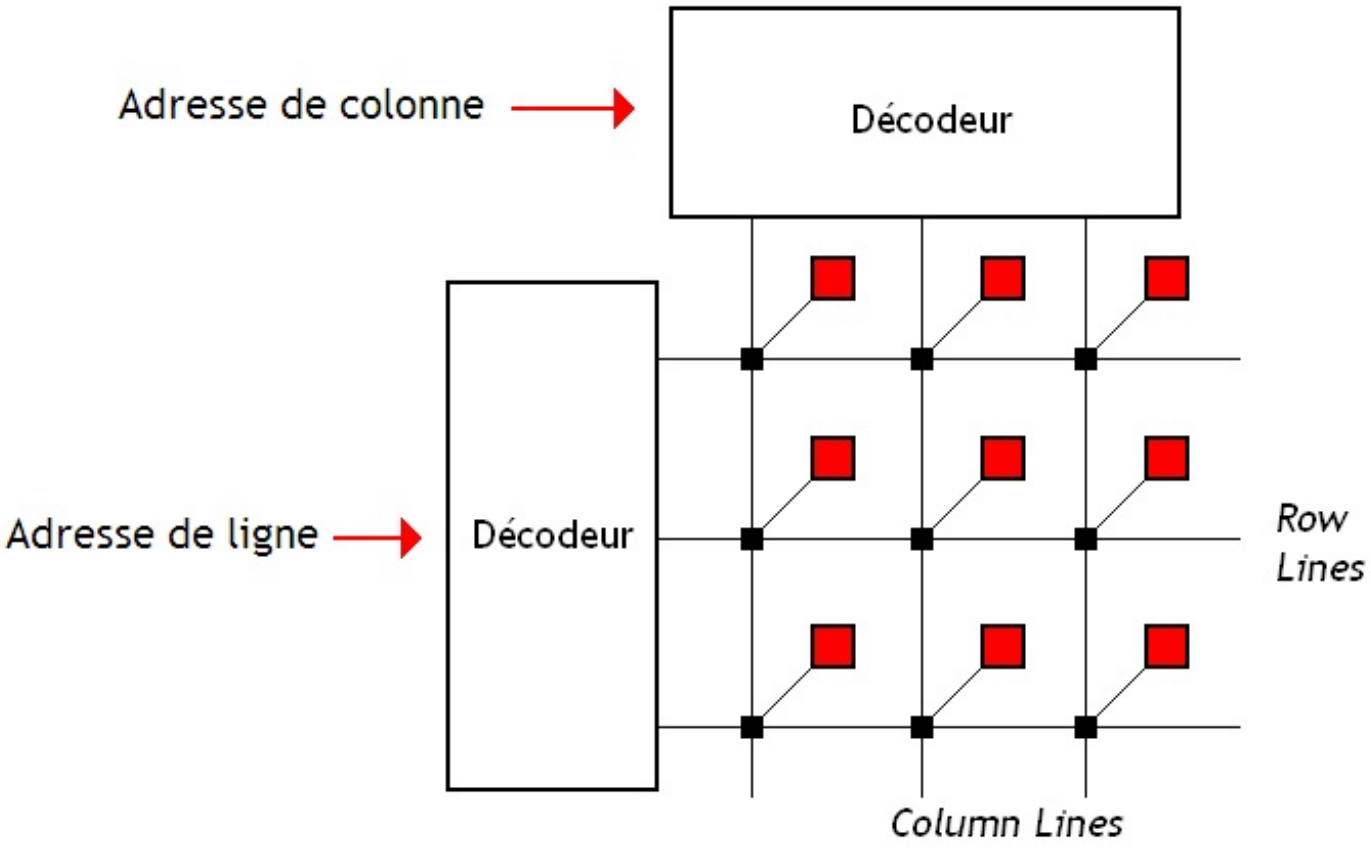
Il faut préciser qu'on trouve une case mémoire à l'intersection entre une colonne et une ligne. Bien évidemment, chaque case mémoire est reliée au bus via un transistor servant d'interrupteur, comme pour les mémoires à adressage linéaire. Et comme pour les mémoires à adressage linéaire, la sélection d'une case mémoire sur le bus se fait grâce à un signal qui ouvrira ou fermera ces fameux transistors. Mais ce signal ne se contentera pas d'une ligne *Row Line*.

Sélectionner une colonne se fera en utilisant un second décodeur. Avec cette solution, toutes les cases mémoires d'une ligne sont reliées à un fil, qu'on nommera le *Row Line*, et toutes les cases mémoires d'une colonne sont reliées à un autre fil : le *Column Line*. Une case mémoire est sélectionnée quand ces deux fils sont mit à 1 : pour cela, il suffit de relier la *Row Line* et la *Column Line* adéquates aux entrées d'une porte ***ET*** dont on relie la sortie sur la grille de notre transistor chargé de relier notre case mémoire au bus.

*Sur ce schéma, les carrés noirs sont les portes **ET** dont je parle au-dessus. Faites attention : les Row Line et les Column Line ne se touchent pas et ne sont pas connectées entre elles : il n'y a pas d'intersection, un des fils passant en-dessous de l'autre.*

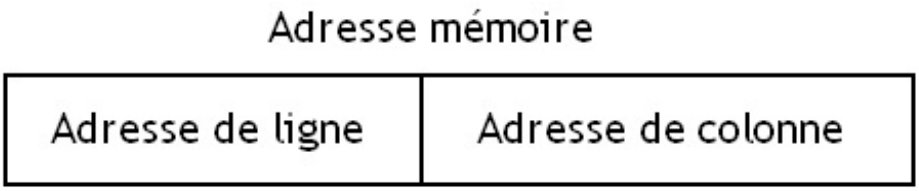


On utilise donc deux décodeurs : un pour sélectionner la *Row Line* reliée à la ligne contenant la case mémoire à lire ou écrire ; et un autre pour sélectionner la colonne.



Adresses hautes et basses

Cela permet de découper notre adresse mémoire en deux morceaux : une adresse haute qui va sélectionner la ligne et une adresse basse qui va sélectionner la colonne.



Influence sur les décodeurs

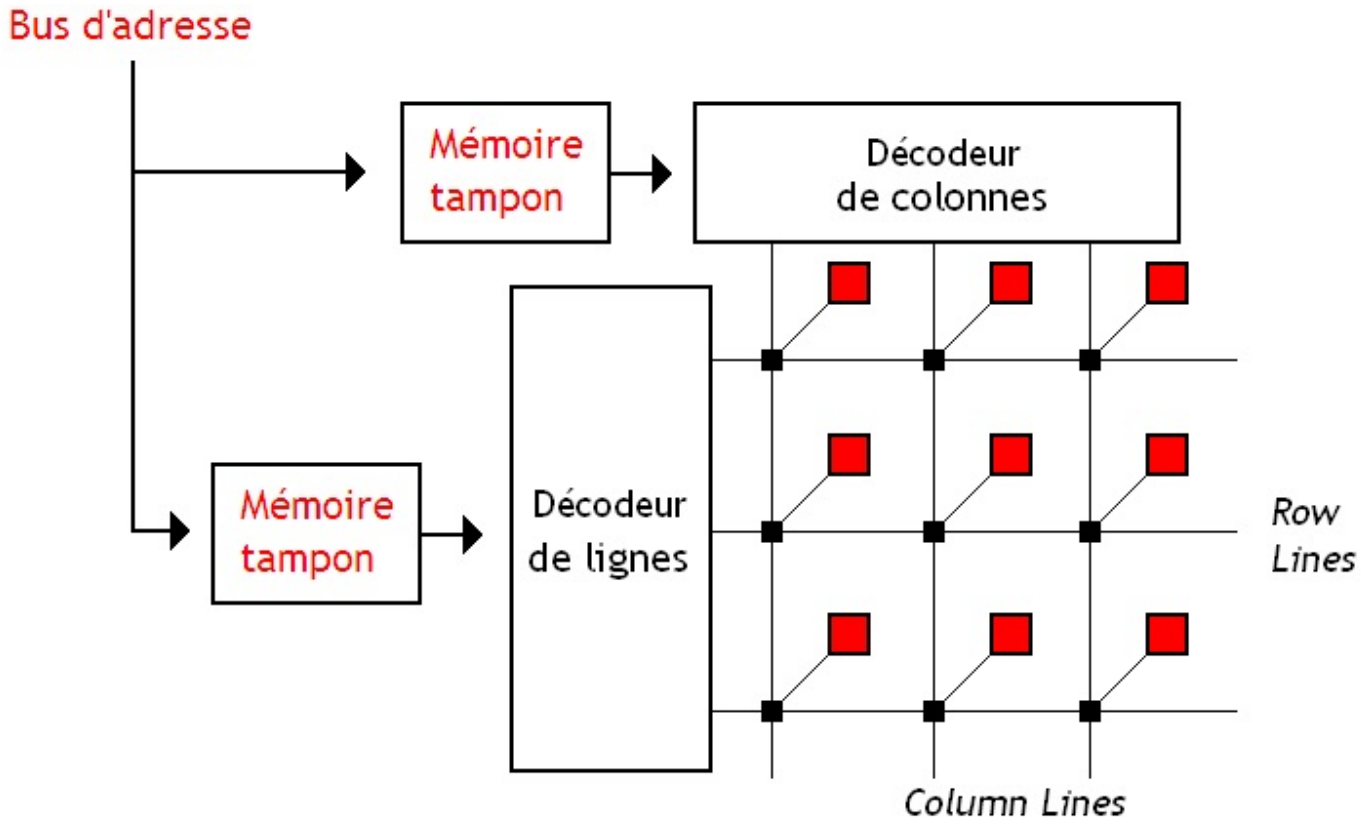
L'adressage par coïncidence a quelques avantages. Le premier de ceux-ci est que nos deux décodeurs peuvent fonctionner en même temps. Ce qui fait qu'on peut décoder une ligne en même temps qu'on décode une colonne. C'est bien plus rapide que l'utilisation d'un seul décodeur, vu que les décodeurs de lignes et de colonnes sont plus petits et plus rapides.

Autre avantage : on économise beaucoup de portes logiques. Il faut savoir que le nombre de portes logiques dépend fortement du nombre d'entrée : moins un décodeur a d'entrée, mieux c'est ! Utiliser deux décodeurs ayant un faible nombre d'entrée est donc plus économique qu'utiliser un gros décodeur avec deux fois plus d'entrées. L'économie en transistors n'est vraiment pas négligeable.

Double envoi

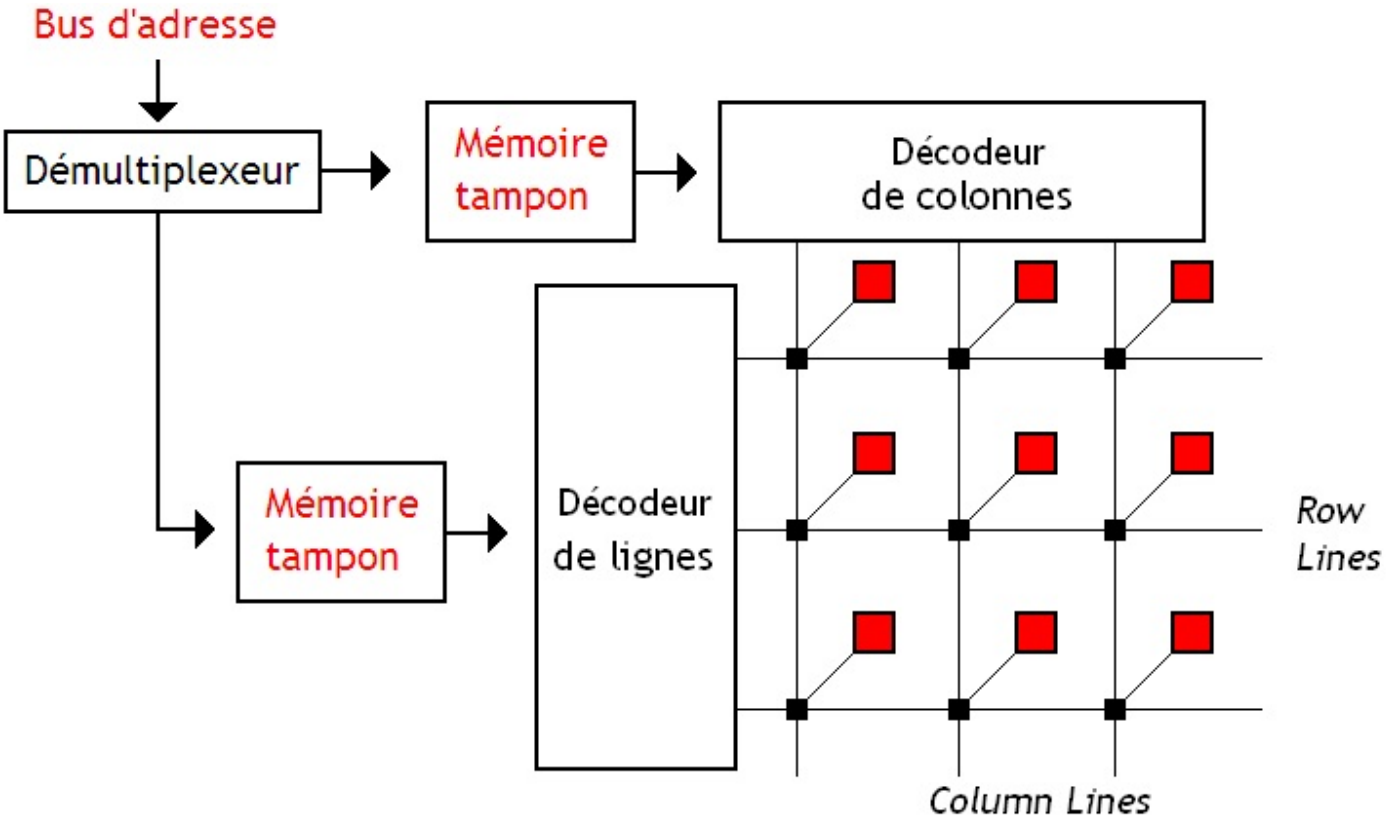
Cela peut permettre d'envoyer nos adresses en deux fois au lieu d'une : on envoie d'abord l'adresse de ligne, puis l'adresse de colonne. Ainsi, on peut économiser des broches et diminuer le nombre de fils pour le bus d'adresse. Vu que ces broches ont un cout assez important, c'est un bénéfice pas négligeable. Mais cela nécessite de modifier l'intérieur de notre mémoire.

Si notre adresse est envoyée en deux fois, notre mémoire doit mémoriser les deux morceaux de l'adresse. Si notre mémoire ne se souvient pas de l'adresse de la ligne, envoyée en premier, elle ne pourra pas sélectionner le byte voulu. On doit donc rajouter de quoi mémoriser l'adresse de la ligne, et l'adresse de la colonne. Pour cela, on place deux mémoires tampons, deux registres, entre les décodeurs et le bus d'adresse.



Ceci dit, ajouter ces mémoires tampons ne suffit pas. Si on envoie l'adresse d'une ligne sur le bus d'adresse, celle-ci doit être

recopiée dans la mémoire tampon située avant le décodeur d'adresse. Et réciproquement avec les colonnes. Il faut donc ajouter de quoi aiguiller le contenu du bus d'adresse vers le bon registre, la bonne mémoire tampon. Cela se fait en utilisant un composant élèctronique nommé un démultiplexeur.



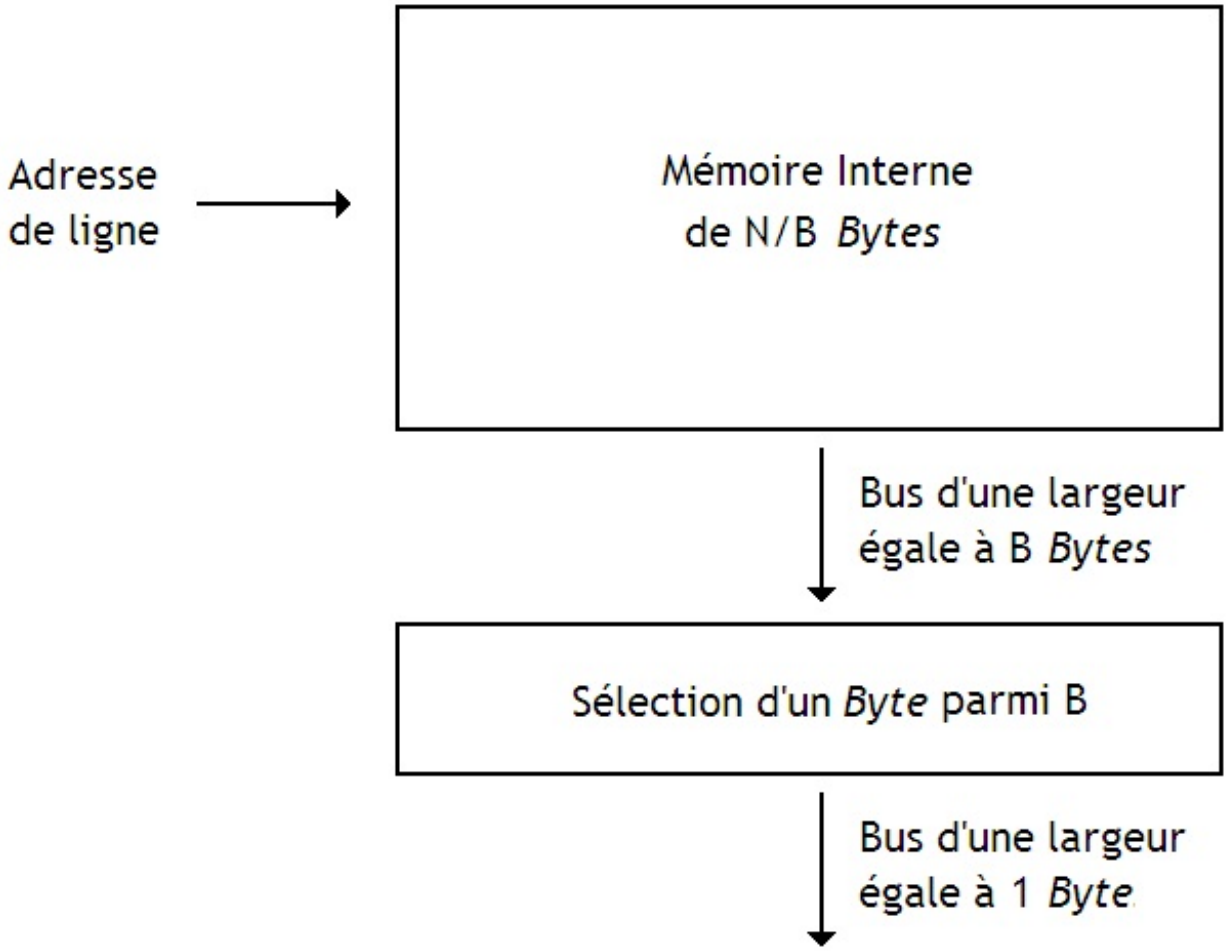
Mémoire à Row Buffer

Enfin, nous allons voir une espèce de mélange entre les deux types de mémoires vues précédemment : les mémoires à ***Row Buffer***. Ces mémoires utilisent le même principe que les mémoires à adressage par coïncidence : elles regroupent plusieurs cases mémoires sur une seule ligne. Mais la sélection de la colonne s'effectue différemment.

Principe

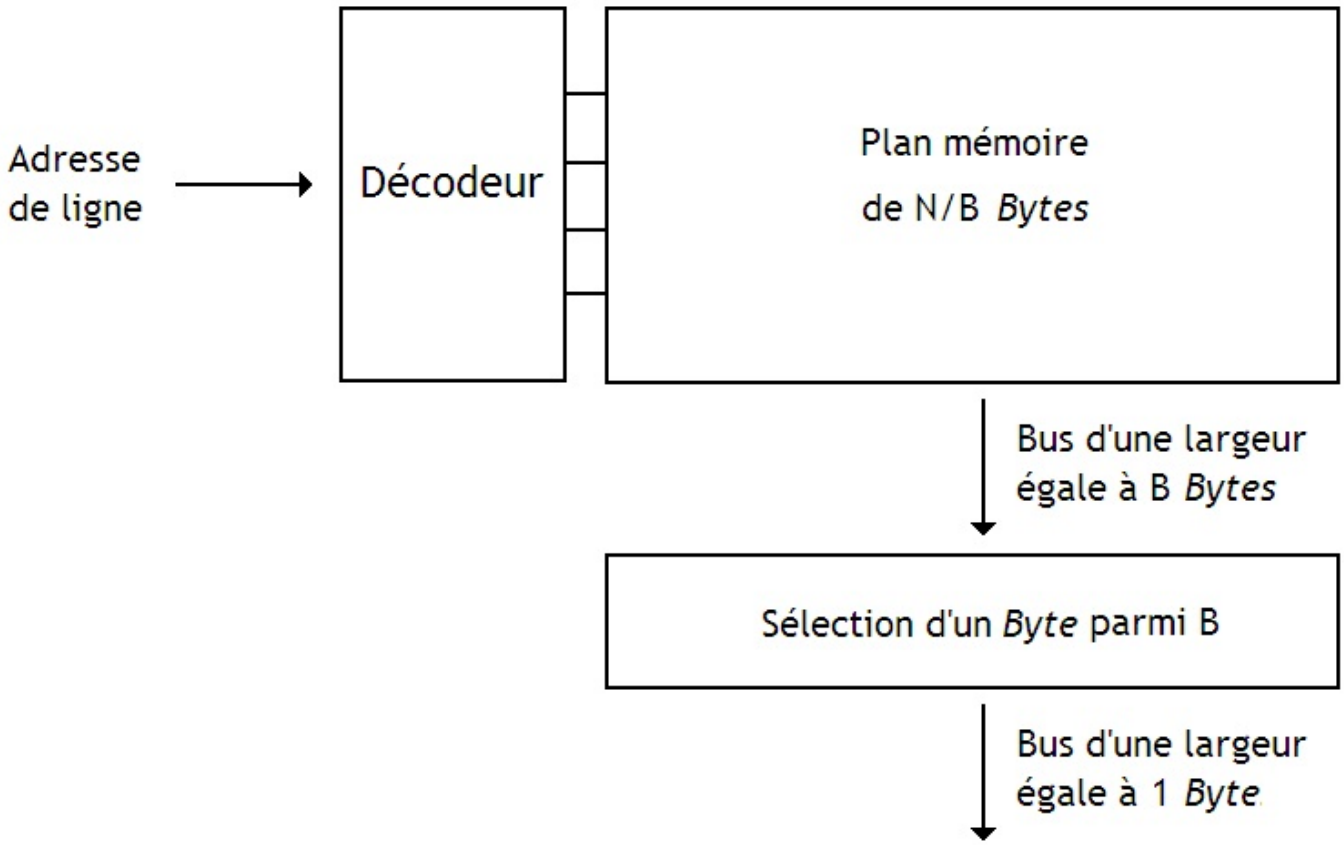
Ces mémoires ont toujours pour but de diminuer le nombre de lignes présentes dans la mémoire. Et les raisons sont les mêmes : éviter de se retrouver avec un décodeur énorme et lent, diminuer la longueur des *Bit Lines*, simplifier la conception de la puce, etc. Pour diminuer ce nombre de lignes, les mémoires à *Row Buffer* vont regrouper plusieurs Bytes sur une même ligne, et sélectionner le bon *Byte* suivant les besoins. C'est le même principe que pour les mémoires à adressage par coïncidence, du moins à première vue.

Une mémoire à *Row Buffer* va émuler une mémoire de N *Bytes* à partir d'une mémoire contenant moins de *Bytes*. Seule différence, ces Bytes internes à la mémoire seront plus gros, et rassembleront plusieurs Bytes externes. Ainsi, ma mémoire à *Row Buffer* sera constituée d'une mémoire interne, contenant B fois moins de *Bytes*, mais dont chacun des *Bytes* seront B fois plus gros. Lorsqu'on veut lire ou écrire dans une mémoire à *Row Buffer*, on va lire un " Super-Byte " de la mémoire interne, et on va sélectionner le bon Byte dans celui-ci.



Plan mémoire

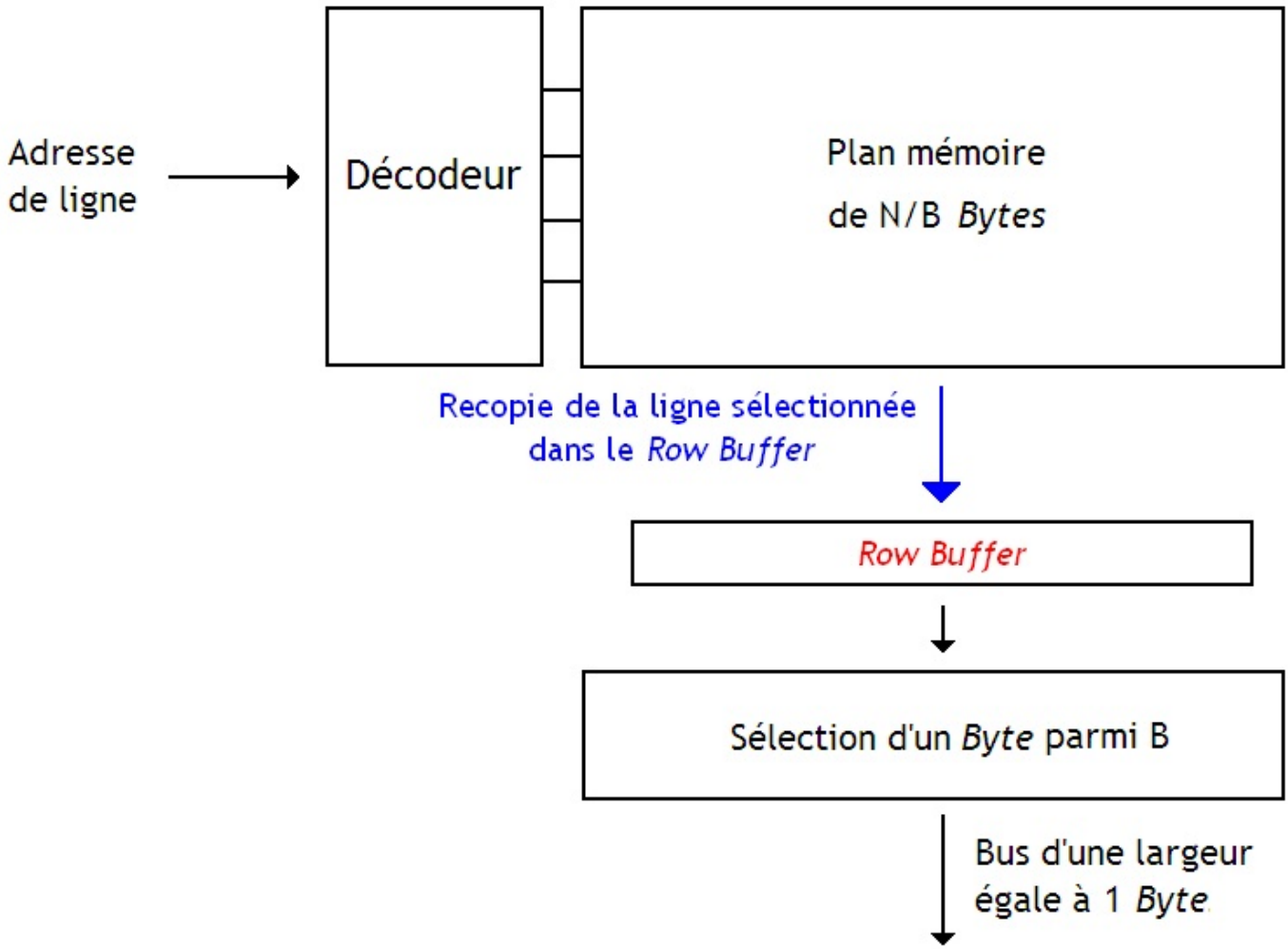
Sur le schéma du dessus, on voit bien que notre mémoire est composée de deux grands morceaux : une mémoire interne, et un circuit de sélection d'un *Byte* parmi B. La mémoire interne n'a rien de spécial : il s'agit d'une mémoire à adressage linéaire tout ce qu'il y a de plus classique. Elles n'utilisent qu'un seul décodeur, qui ne sert qu'à sélectionner la ligne.



Par contre, le circuit de sélection est un nouveau venu, tout comme l'interface entre la mémoire interne et ce circuit.

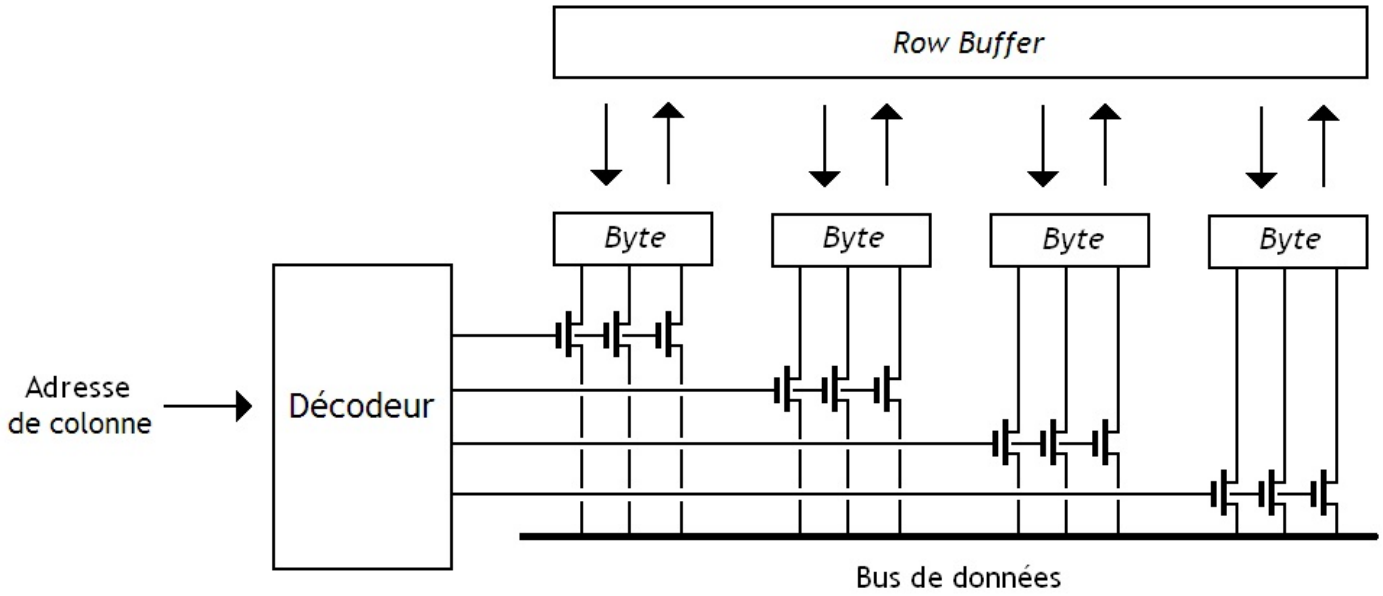
Row Buffer

Je me permets de signaler que sur la grosse majorité de ces mémoires, la mémoire interne n'est pas reliée directement sur ce circuit de sélection des colonnes. Chaque ligne sélectionnée dans notre mémoire interne est recopiée intégralement dans une sorte de gros registre temporaire, dans lequel on viendra sélectionner la case mémoire correspondant à notre colonne. Ce gros registre temporaire s'appelle le ***Row Buffer***.



Sélection de colonnes

Dans sa version la plus simple, le circuit de sélection de colonnes est implémenté avec un décodeur. Il est donc composé d'un décodeur, et de séries de transistors, qui vont (ou non) relier un *Byte* au bus.



Il y a d'autres façons d'implémenter ce circuit de sélection de colonnes, mais on n'en parlera pas ici. Il faut dire que ces autres méthodes sont plus lentes : pour une fois, la méthode simple est la plus efficace.

Avantages et inconvénients

Les plans mémoire à *Row Buffer* récupèrent les avantages des plans mémoires par coïncidence : possibilité de décoder une ligne en même temps qu'une colonne, possibilité d'envoyer l'adresse en deux fois, consommation moindre de portes logiques, etc.

Autre avantage : en concevant correctement la mémoire, il est possible d'améliorer les performances lors de l'accès à des données proches en mémoire : si on doit lire ou écrire deux Bytes localisés dans la même ligne de notre mémoire interne, il suffit de charger celle-ci une fois dans le *Row Buffer*, et de faire deux sélections de colonnes différentes. C'est plus rapide que de devoir faire deux sélection de lignes et deux de colonnes. On en reparlera lorsqu'on verra les mémoires SDRAM et EDO, vous verrez.

De plus, cela permet d'effectuer l'opération de rafraichissement très simplement. Il suffit de recopier le contenu d'une ligne dans le *Row Buffer*, avant de faire l'inverse en recopiant le contenu du *Row Buffer* dans la ligne mémoire sélectionnée. Il suffira d'intercaler un circuit chargé du rafraichissement quelque part dans notre contrôleur mémoire pour que le tout fonctionne.

Pas contre, cette organisation a un défaut : elle consomme beaucoup d'énergie. Il faut dire que pour chaque lecture d'un Byte dans notre mémoire, on doit charger une ligne de la mémoire interne dans le *Row Buffer*, qui contient plusieurs Bytes. Et cela pompe du courant de recopier tous ces Bytes dans le *Row Buffer*. Ce n'est pas le cas avec une mémoire à adressage linéaire ou par coïncidence : on ne sélectionne que le *Byte* que l'on veut lire ou écrire.

Interfacage avec le bus

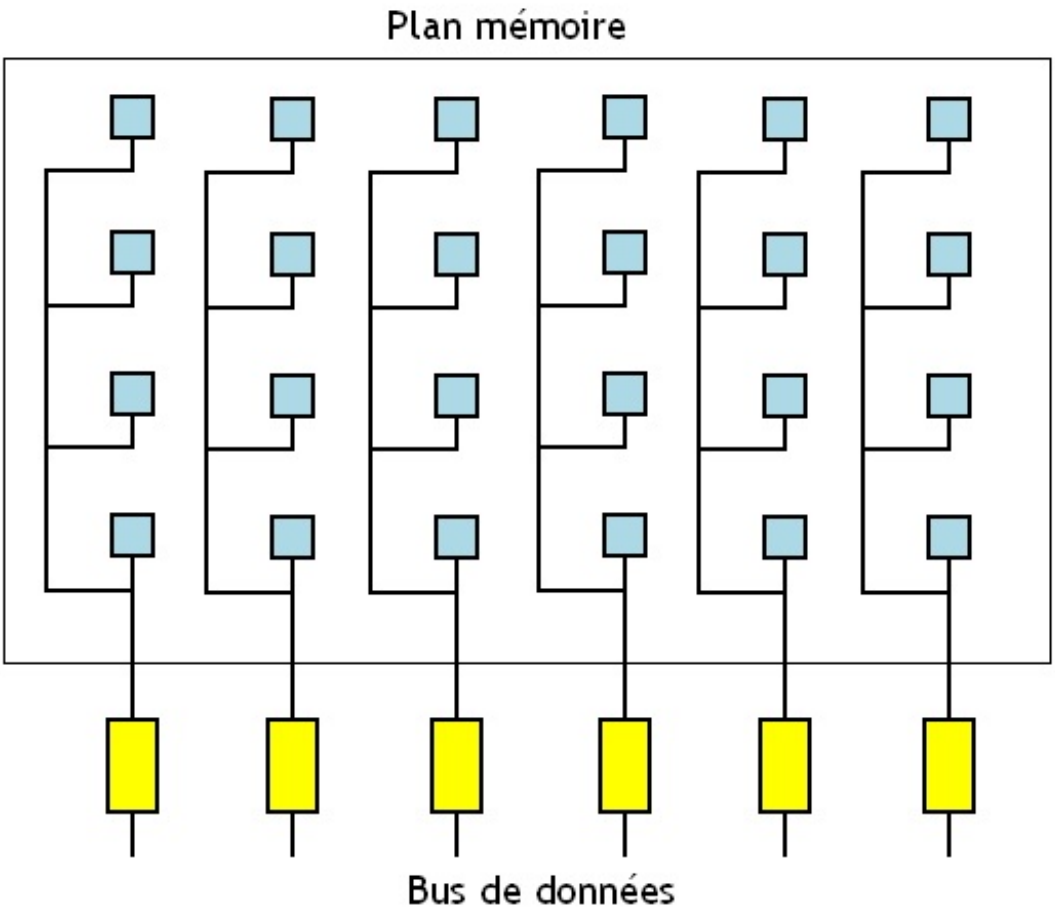
Avec ce qu'on a vu plus haut, on sait comment adresser une case mémoire, mais il nous reste une chose à faire : on peut lire ou écrire dans une case mémoire et il nous faut donc commander nos circuits de façon à ce qu'ils fassent une lecture ou une écriture. Dans le cas d'une lecture, le contenu de notre case mémoire est recopiée sur le bus. Dans le cas d'une écriture, c'est l'inverse : le contenu du bus qui est recopié dans la case mémoire sélectionnée. Suivant qu'on fasse une lecture ou écriture, le sens de transfert des données n'est pas le même.

Comment choisir le sens de transfert des données ?

Circuits 3-états

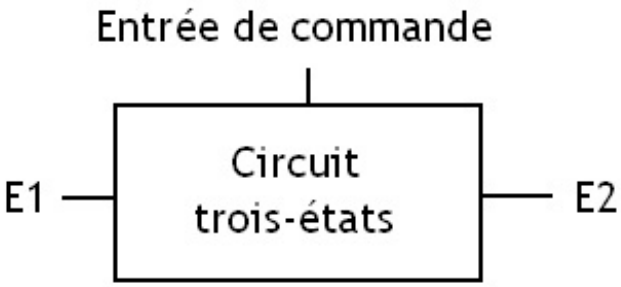
Première solution : comme à chaque fois qu'on a un problème, on rajoute un circuit. Il suffit simplement d'intercaler des composants chargés d'imposer le sens des transferts entre le bus de données et les fils reliés aux différentes cases mémoires.

Les circuits chargés d'imposer le sens des transferts de données sont indiqués en jaune sur le schéma suivant.



Ces circuits sont ce qu'on appelle des **circuits 3-états**. Rien de bien méchant, il s'agit juste de circuits électroniques fabriqués avec des transistors qui vont apporter une solution à notre problème.

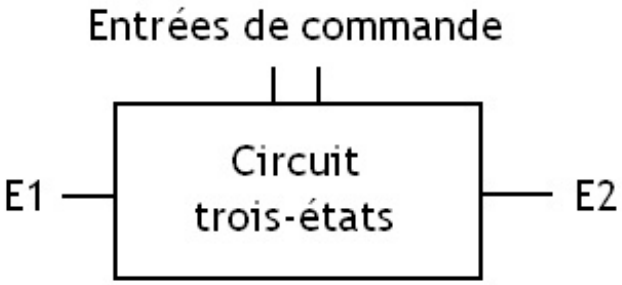
Voici à quoi ressemble un circuit 3-états :



Notre circuit 3-états permet de préciser le sens de transfert : soit de E1 vers E2, soit l'inverse. Il peut aussi se comporter comme un interrupteur ouvert, et ainsi déconnecter les deux fils. Le choix du mode de fonctionnement se fera par ce qu'on mettra sur son entrée de commande :

- soit le circuit 3-états agit comme un interrupteur ouvert ;
- soit le circuit 3-états recopie la tension présente sur l'entrée E1 sur l'entrée E2 : le bit envoyé sur E1 va alors passer vers le fil relié à E2 ;
- soit circuit 3-états recopie la tension présente sur l'entrée E2 sur l'entrée E1 : le bit envoyé sur E1 va alors passer vers le fil relié à E1.

On a besoin de préciser trois cas, donc on utilise deux entrées pour cela.

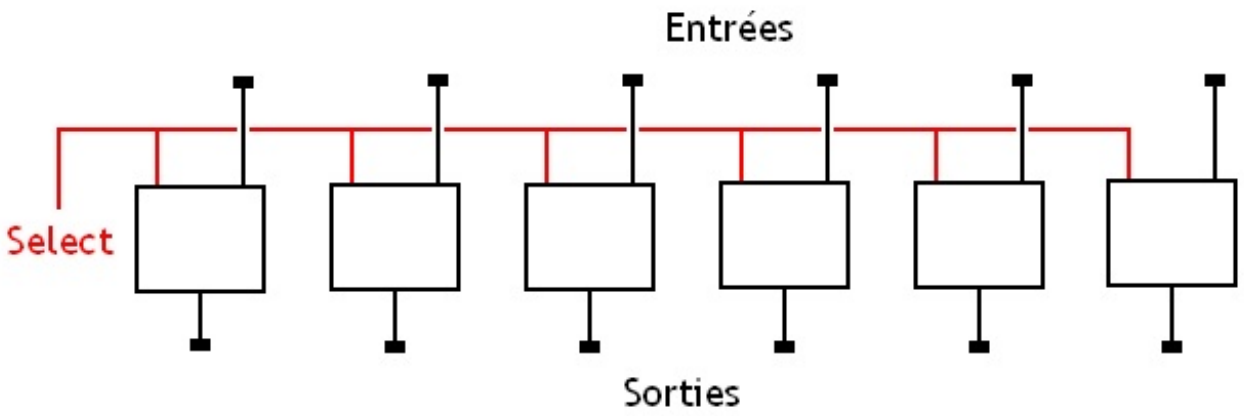


On a donc résolu notre problème : il suffit d'intercaler un circuit 3-états entre le plan mémoire et le bus, et de commander celui-ci correctement. C'est le contrôleur mémoire qui place l'entrée de commande du circuit 3-états pour sélectionner le sens de transfert de donnée suivant l'état du bit R/W.

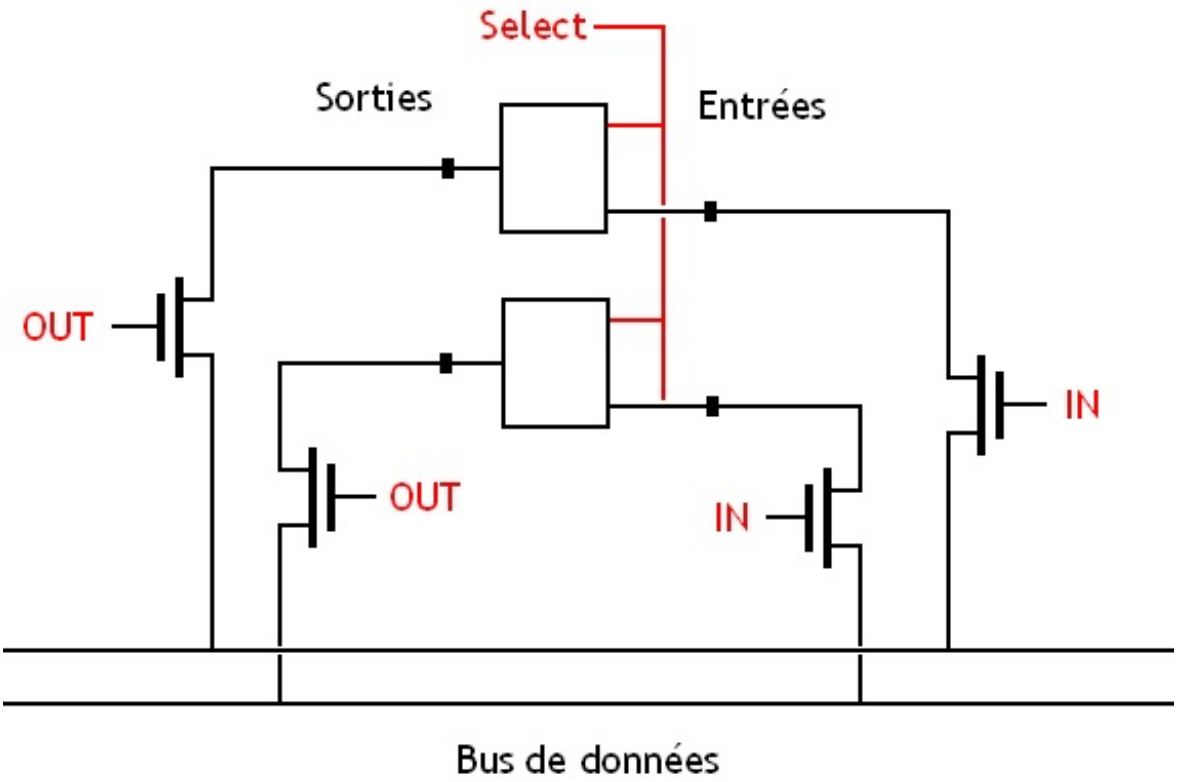
Petit détail : ce circuit 3-états ne réagit pas immédiatement à un changement de son entrée. En clair : il met un certain temps avant de passer d'une lecture à une écriture (et vice-versa). En clair, effectuer des lectures les unes après les autres sera plus rapide qu'alterner lectures et écritures. Ce temps d'attente n'est pas toujours négligeable, et il faut parfois le prendre en compte. Souvenez-vous : on en reparlera au prochain chapitre.

Mémoires à ports de lecture et écriture séparés

Encore une fois, la solution vue plus haut n'est pas la seule. Su d'autres mémoires, on n'a pas besoin d'utiliser des circuits trois états. Sur ces mémoires, la lecture et l'écriture ne passe pas par les mêmes fils. C'est le cas pour les registres. Souvenez-vous à quoi ressemble un registre.



Comme vous le voyez, écrire une donnée ne passe pas par les mêmes fils que la lecture. Ainsi, la solution est évidente : il suffit de relier tout ces fils sur le bus, et connecter les bons fils pour choisir le sens de transfert.

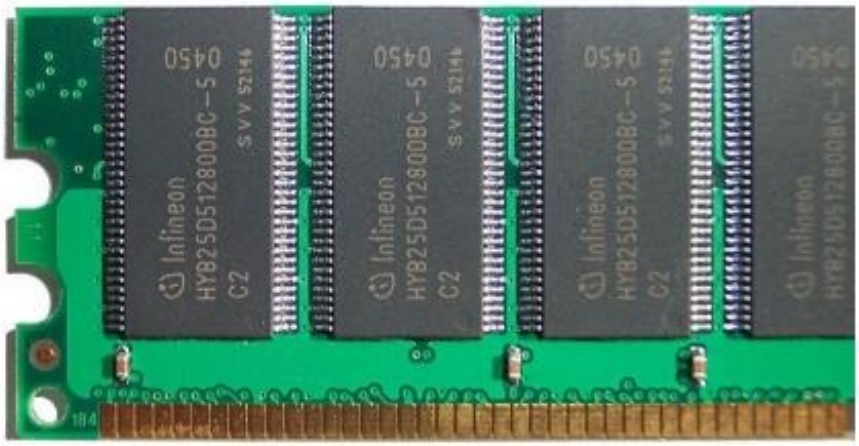


Comme vous le voyez sur ce schéma, les fils dédiés à la lecture ou l'écriture sont tous reliés sur le bus, avec un transistor intercalé au bon endroit. Il suffit d'ouvrir ou de fermer les bons transistors pour déclencher soit une lecture (on ferme les transistors IN) ou une écriture (on ferme les transistors OUT). C'est le contrôleur mémoire qui se charge de déduire les tensions IN et OUT à envoyer sur les grilles des transistors intercalés entre le bus et la case mémoire.

Assemblages de mémoires

Ce qu'on a vu plus haut est l'organisation interne d'une mémoire RAM/ROM simple. Dans la réalité, il faut savoir que nos mémoires ne sont pas faites d'un seul bloc, mais sont constituées de plusieurs mémoires plus simples rassemblées dans un seul boitier.

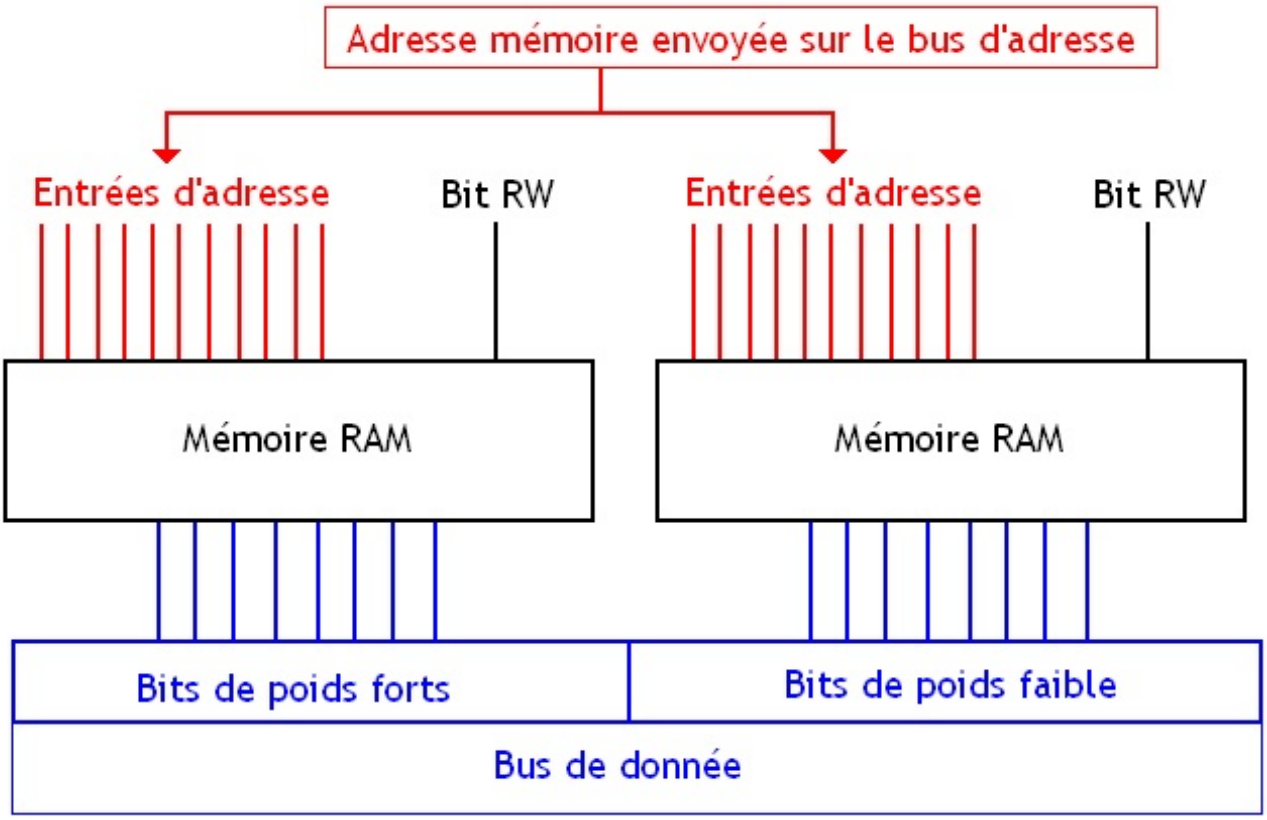
Si vous ne me croyez pas, prenez une barrette de mémoire RAM, et regardez de plus prêt.



Vous voyez : chaque puce noire sur notre barrette de mémoire RAM est une petite mémoire à elle toute seule. Sur une barrette de mémoire RAM, on trouve pleins de petites mémoires rassemblées sur un seul circuit imprimé (notre barrette de mémoire) et organisées d'une certaine façon. Mais avant de passer à la pratique, une petite précision s'impose : rassembler plusieurs mémoires dans un seul boitier peut être fait de diverses façons différentes : on peut décider de doubler le nombre d'adresses, doubler la taille d'un *byte*, ou faire les deux. Suivant ce que l'on veut faire, l'organisation de nos sous-mémoires ne sera pas la même.

Arrangement horizontal

Commençons par notre premier cas : on va utiliser plusieurs boitiers pour doubler la taille d'un *byte* sans changer le nombre de cases mémoires adressables. C'est ce qu'on appelle l'**arrangement horizontal**. Prenons un exemple : on va cherche à obtenir une mémoire ayant un *byte* de 16 bits en partant de deux mémoires ayant un *byte* de 8 bits. Pour cela, il suffit de relier nos mémoires de façon à ce que la première contienne les bits de poids fort du byte de 16 bits, et que l'autre contienne les bits de poids faible.



On voit que l'on adresse plusieurs mémoires en même temps : chaque mémoire contient un morceau de la donnée qu'on souhaite lire ou écrire. Il suffit de les relier sur le bus correctement, de façon à ce que chaque morceau de la donnée aille au bon endroit. Évidemment, chaque morceau de donnée possède la même adresse que les autres morceaux : l'adresse (et les entrées de commande : difficile de lire un morceau de case mémoire perdant qu'on en écrit un autre) est envoyée sur toutes les sous-mémoires.

Mémoires à plusieurs plans mémoires

Avec cette organisation, tout se passe comme si notre mémoire était composée de plusieurs plans mémoires : un par boitier. Ce qu'on vient de voir peut vous semblez d'une étonnante simplicité, mais sachez que c'est quelque chose de similaire qui est utilisé dans nos barrettes de mémoires actuelles. Sur les mémoires SDRAM ou DDR-RAM présentes à l'intérieur de notre PC, on utilise beaucoup cet arrangement horizontal.

Toutes ces mémoires possèdent un *byte* de 8 bits, mais sont en réalité composées de 8 sous-mémoires ayant un *byte* de 1 bit. Ces 8 sous-mémoires correspondent aux puces noires présentes sur vos barrettes de mémoires RAM.

L'intérêt de faire comme ceci ? Maitriser la répartition de chaleur à l'intérieur de notre mémoire. Accéder à une cellule mémoire, ça chauffe ! Si on utilisait une seule puce mémoire ayant un *byte* de 8 bits, les 8 bits auxquels on accèderait seraient placés les uns à coté des autres et chaufferaient tous au même endroit. A la place, les fabricants de mémoire RAM préfèrent disperser les bits d'un même octet dans des puces différentes pour répartir la chaleur sur une plus grande surface et la disperser plus facilement. Sans cela, nos mémoires ne fonctionneraient tout simplement pas sans systèmes de refroidissements adaptés.

Dual channel

Vous avez surement déjà entendu parler de *dual-channel*. Et bien sachez que cette technologie est basée sur le même principe. Sauf qu'au lieu de rassembler plusieurs puces mémoires (les trucs noirs sur nos barrettes) sur une même barrette, on fait la même chose avec plusieurs barrettes de mémoires. Ainsi, on peut mettre deux barrettes ayant un bus de donnée capable de contenir 64 bits et on les relie à un bus de 128 bits. C'est ce qu'on appelle le *dual-channel*.

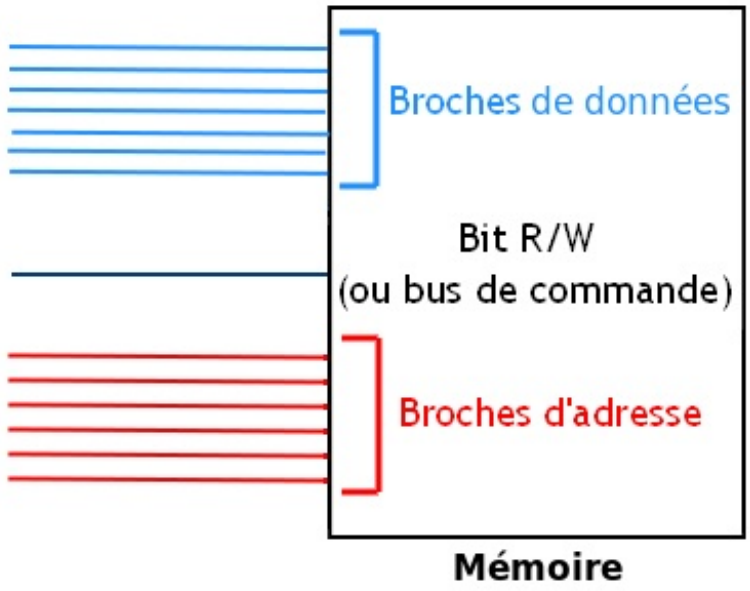
Arrangement vertical

Une autre possibilité consiste à rassembler plusieurs boitiers de mémoires pour augmenter la capacité totale. On utilisera un boitier pour une partie de la mémoire, un autre boitier pour une autre, et ainsi de suite. Par exemple, on peut décider d'utiliser deux sous-mémoires : chacune de ces sous-mémoire contenant la moitié de la mémoire totale. C'est ce qu'on appelle l'**arrangement vertical**.

Ça peut vous paraitre bizarre, mais réfléchissez un petit peu, vous pouvez trouver un exemple parfait d'arrangement vertical dans votre PC : vous pouvez parfaitement placer plusieurs barrettes de mémoires sur votre carte mère pour doubler la capacité. Si vus n'utilisez pas de *dual-channel*, vous aurez un arrangement vertical.

Entrée de sélection

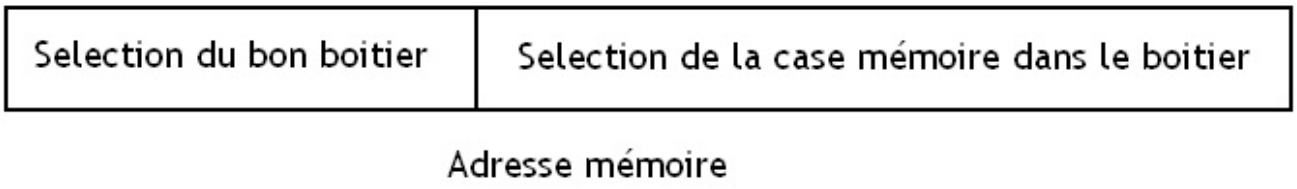
Avec cette solution, quand on accède à une case mémoire de notre grosse mémoire rassemblant tous les boitiers, on accède à un boitier parmi tout les autres. Il nous faudra sélectionner le bon boitier, celui qui contient notre case mémoire. Pour cela, il va falloir que je vous dise quelque chose. Vous vous souvenez que dans le chapitre précédent, j'ai dit qu'une mémoire ressemblait à ça :



Mais j'ai volontairement passé un détail sous silence : nos mémoires possèdent une broche supplémentaire nommée CS, qui sert à activer ou désactiver la mémoire. Suivant la valeur de ce bit, notre mémoire sera soit placée en veille sans possibilité de réagir aux évènements extérieurs (elle conservera toutefois son contenu), soit fonctionnera normalement. Et c'est grâce à cette broche qu'on va pouvoir rassembler plusieurs boitiers dans notre mémoire.

Une histoire d'adresse

Avec cette organisation, une adresse mémoire est découpée en deux partie : une partie qui est décodée pour sélectionner le bon boitier, et une autre qui sert à sélectionner la case mémoire dans le boitier. Sur certaines mémoires, on utilise les bits de poids forts pour sélectionner le bon boitier.

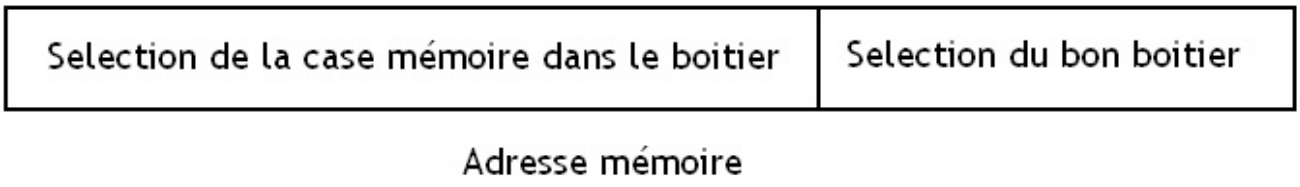


l'idée est simple : pour accéder à une case mémoire, on envoie le morceau de l'adresse sélectionnant la case mémoire dans le boitier et le bit R/W à toutes les mémoires, et on éteint les mémoires qui ne stockent pas la case mémoire voulue. Seul le boitier contenant la donnée restera allumée et pourra répondre à la commande (adresse + ordre de lecture/écriture) envoyé par le contrôleur mémoire.

Il suffira donc de positionner les bits CS de chaque mémoire à la bonne valeur. Un seul boitier sera allumé et aura donc son bit CS à 1, tandis que tous les autres seront éteints et verront leur bit CS passer à 0. Le fait que seul un boitier parmi tous les autres soit activé devrait vous rappeler quelque chose : oui, on utilise bien un décodeur pour sélectionner le boitier.

Mémoires interleaved

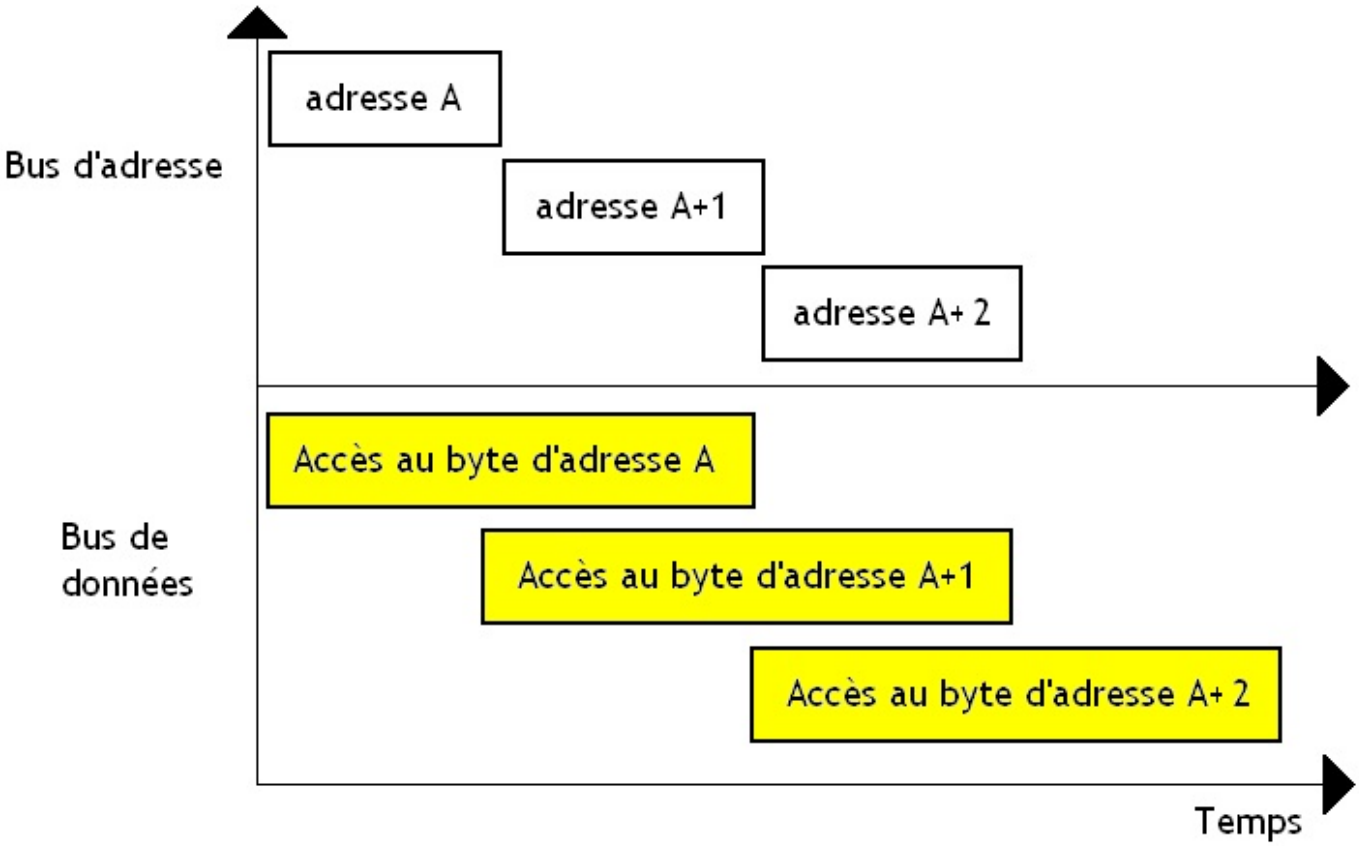
Mais certaines mémoires, on utilise les bits de poids faible pour sélectionner le bon boitier : ce sont ce qu'on appelle des **mémoires *interleaved***.



En faisant ainsi, on peut mettre des *bytes* consécutifs dans des mémoires différentes. Cela permet, en adaptant le contrôleur mémoire, d'obtenir des opportunités d'optimisation assez impressionnantes. Et là, je suis sûr que vous ne voyez pas vraiment pourquoi. Aussi, une explication s'impose. Il faut savoir dans la grosse majorité des cas, les accès mémoires se font sur des bytes consécutifs : les lectures ou écritures dans la mémoire se font souvent par gros blocs de plusieurs *bytes*. Il y a diverses raisons à cela : l'utilisation de tableaux par les programmeurs, le fait que nos instructions soient placées les unes à la suite des autres dans la mémoire, etc. Quoiqu'il en soit, l'accès à des zones de mémoire consécutives est quelque cchose que l'on doit optimiser le plus possible.

Mais l''accès à un boitier prend toujours un peu de temps : c'est le fameux temps d'accès dont on a parlé il y a quelques chapitres. Si on place deux *bytes* ayant des adresses consécutives dans le même boitier, et qu'on souhaite lire/écrire ces deux bytes, on devra attendre que l'accès au premier *byte* soit fini avant de pouvoir accéder au suivant (sauf si la mémoire est multiports, mais bref). En clair : on ne peut effectuer qu'un seul accès à la fois sur des *bytes* consécutifs.

Mais ce n'est valable qu'avec les mémoires qui ne sont pas des mémoires *interleaved* ! Avec les mémoires *interleaved*, la donne est différentes : des bytes consécutifs sont localisés dans des boitiers différentes qui peuvent être accédés en parallèle. On peut ainsi accéder à des bits consécutifs bien plus rapidement qu'avec des mémoires "normales".



Et voilà, maintenant que vous avez lu ce chapitre, vous êtes capables de créer une petite mémoire RAM assez simples. Ça vous plait ?

Mémoires DDR, SDRAM et leurs cousines

Au chapitre précédent, on a vu comment des mémoires RAM ou ROM simples étaient organisées. Mais les mémoires actuelles sont un peux plus complexes que les mémoires simplistes qu'on a vues au chapitre précédent. Ce chapitre va vous expliquer dans les grandes lignes en quoi nos mémoires actuelles sont meilleures, et comment elles fonctionnent. Rassurez-vous, vous serez tout de même en terrain connu : les mémoires RAM actuelles ne sont que des améliorations des mémoires vues précédemment.

Les mémoires RAM asynchrones

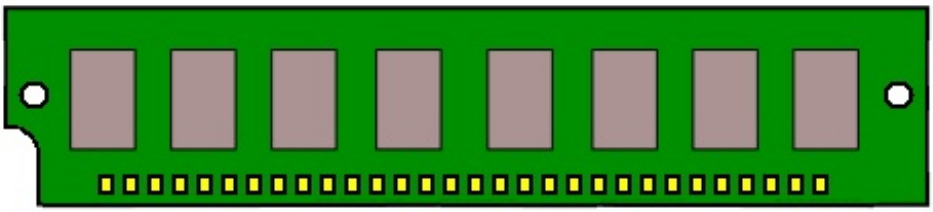
De nos jours, vos mémoires sont des mémoires DDR1, DDR2, voire DDR3. Mais avant l'invention de ces DDR, il a existé un grand nombre de mémoires plus ou moins différentes. Vous connaissez surement les mémoires SDRAM, mais on va commencer par encore plus ancien : nous allons parler des mémoires utilisées sur les premiers processeurs Intel comme les 486 DX et le premier Pentium, à savoir la mémoire FPM et la mémoire EDO. La première mémoire évoluée qui fut inventée s'appelle la mémoire FPM, ou ***Fast Page Mode***. Ce fut la première mémoire à être produite sous la forme de barrettes. Elle fut suivi quelques années plus tard de la mémoire EDO-RAM, qui n'en est qu'une amélioration.

Ces mémoires étaient toutes des mémoires qui n'étaient pas synchronisées avec le processeur via une horloge. On appelle de telles mémoires des **mémoires asynchrones**. Quand ces mémoires ont étés crées, cela ne posait aucun problème : la mémoire était tellement rapide que le processeur n'avait pas vraiment à se synchroniser avec la mémoire. Une lecture ou écriture prenait nettement moins de temps qu'un cycle d'horloge, et le processeur était certain que la mémoire aurait déjà fini sa lecture ou écriture au cycle suivant. Du moins, c'était vrai au début.

Format des mémoires FPM et EDO

Ces mémoires FPM et EDO-RAM étaient produites sous forme de barrettes qui existaient en deux versions : une version 72 broches, et une version 30 broches. Les broches dont je parle, ce sont les trucs jaunes situées en bas des barrettes de mémoire. Elles servent à connecter les circuits de notre barrette de mémoire sur le bus.

Voici à quoi ressemblait la version 30 broches.



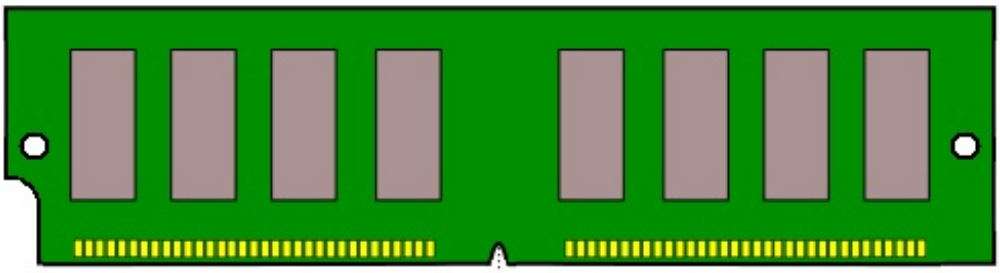
Pour les curieux, voici en détail à quoi servent ces broches.

Détail des broches	Utilité
1	Tension d'alimentation
2	Signal CAS (on en parlera plus tard)
3	Bit 0 du bus de donnée
4	Bit 0 du bus d'adresse
5	Bit 1 du bus d'adresse
6	Bit 1 du bus de données
7	Bit 2 du bus d'adresse
8	Bit 3 du bus d'adresse
9	Masse : zéro volt
10	Bit 2 du bus de données
11	Bit 4 du bus d'adresse
12	Bit 5 du bus d'adresse

13	Bit 3 du bus de données
14	Bit 6 du bus d'adresse
15	Bit 7 du bus d'adresse
16	Bit 4 du bus de données
17	Bit 8 du bus d'adresse
18	Bit 9 du bus d'adresse
19	Bit 10 du bus d'adresse
20	Bit 5 du bus de données
21	Bit R/W
22	Zéro volt : masse
23	Bit 6 du bus de données
24	Bit 11 du bus d'adresse
25	Bit 7 du bus de données
26	Bit de parité pour les données écrites
27	Signal RAS (on en parlera dans ce qui va suivre)
28	Signal CASP
29	Bit de parité pour les données lues
30	Tension d'alimentation (en double)

Si vous vous amusez à compter les nombre de bits pour le bus de donnée et pour le bus d'adresse, vous remarquerez que le bus d'adresse contient 12 bits et que le bus de données en fait 8. Les mémoires 72 broches contiennent plus de bits pour le bus de données : 32 pour être précis. Par contre le bus d'adresse ne change pas : il reste de 12 bits. D'autres bits pour ou moins importants ont étés rajoutés : les bits RAS et CAS sont en plusieurs exemplaires et on trouve 4 fois plus de bits de parité (un par octet transférable sur le bus de données).

Et voilà ce que donnait la version 72 broches.



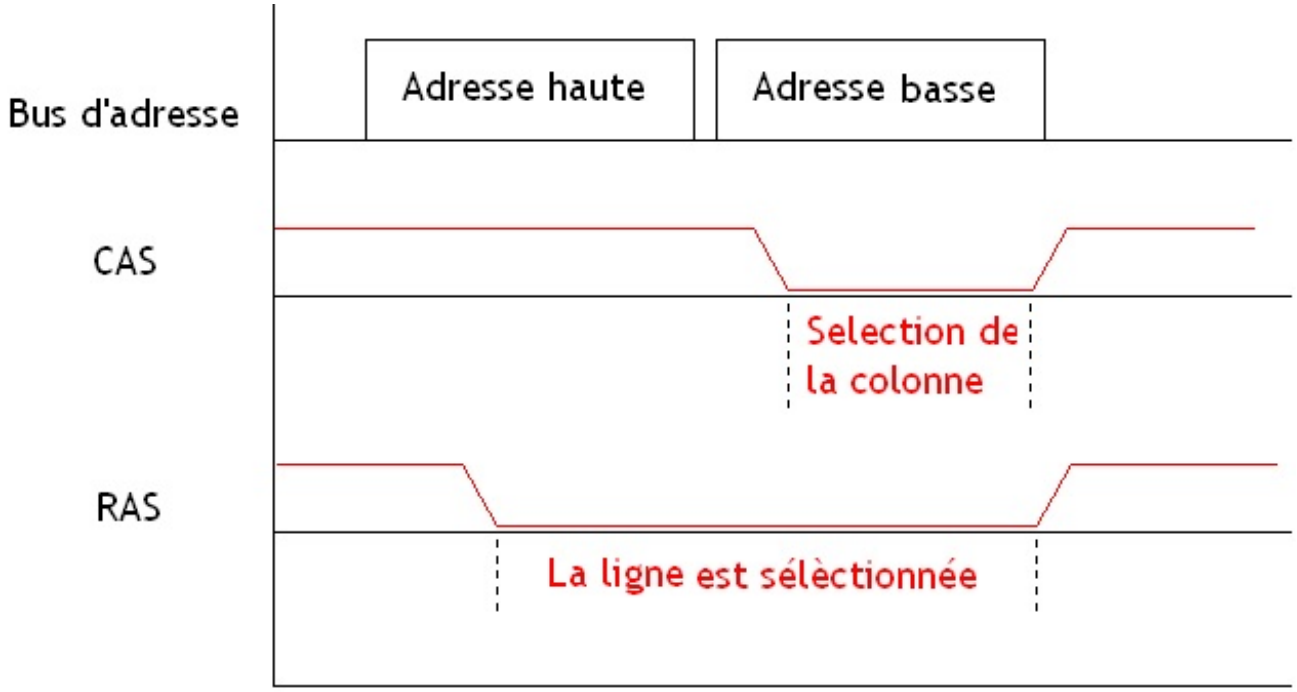
Pour information, la tension d'alimentation des mémoires FPM était de 5 volts. Pour les mémoires EDO, cela variait entre 5 et 3.3 volts.

RAS et CAS

Je ne sais pas si vous avez remarqués, mais le bus d'adresse de 12 bits de ces mémoires ne permettait d'adresser que 4 kibioctets de mémoires, ce qui est très peu comparé à la taille totale des mémoires FPM (qui faisaient plusieurs mébioctets). En fait, il y a une grosse astuce derrière tout ça : l'adresse était envoyée en deux fois. Cette adresse était alors découpée en deux parties : une adresse haute, et une adresse basse. Ces mémoires asynchrones étaient toutes sont des mémoires à adressage par coïncidence ou à *Row Buffer*. Elles étaient donc organisées en lignes et en colonnes. L'adresse haute permettait de sélectionner la ligne du

plan mémoire, et l'adresse basse sélectionnait la colonne.

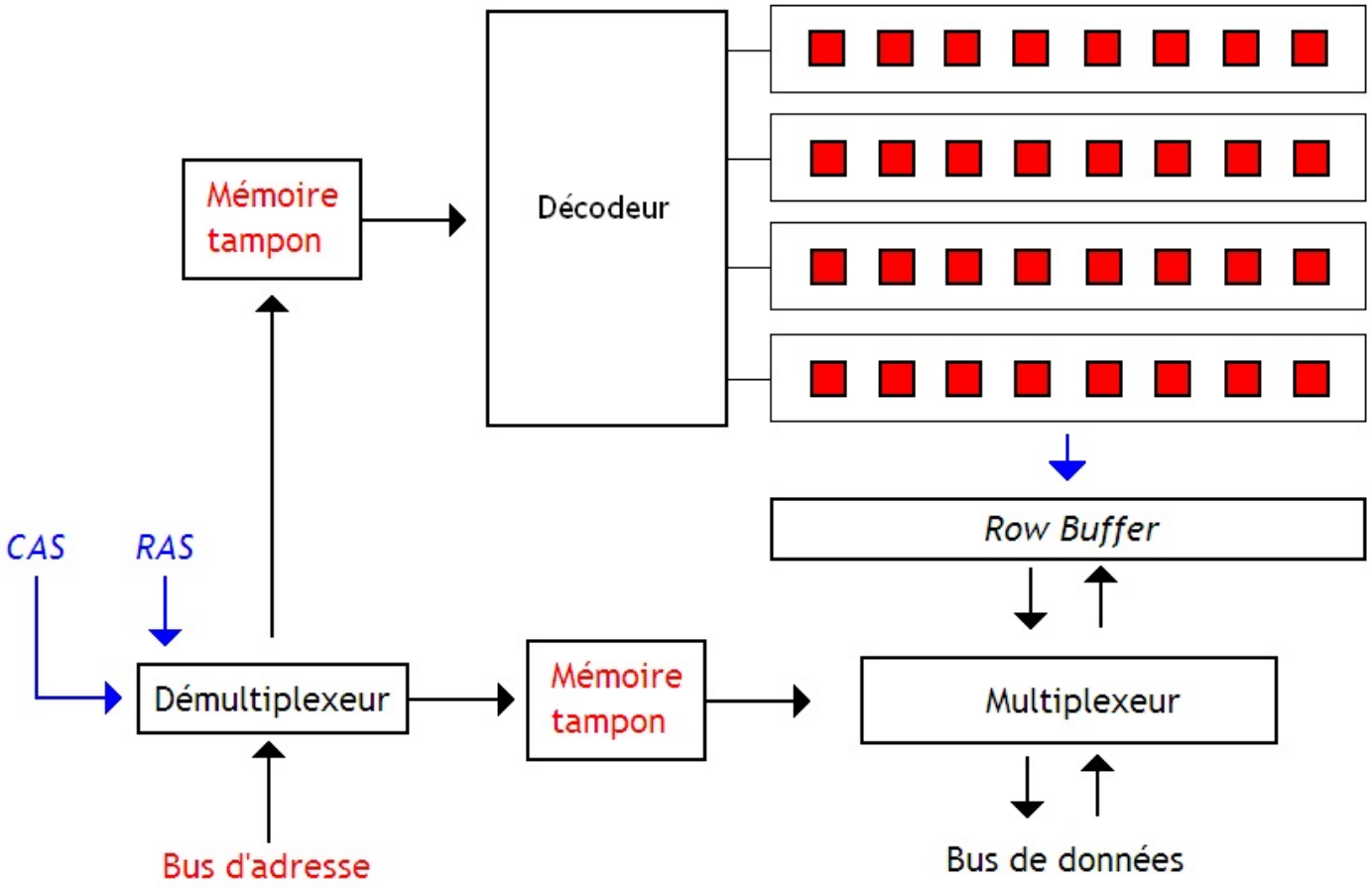
Mais envoyer l'adresse en deux fois nécessitait de dire à la mémoire si le morceau d'adresse présent sur le bus d'adresse servait à sélectionner une ligne ou une colonne. Imaginez ce qui pourrait arriver si jamais la mémoire se trompait ! Pour cela, le bus de commande de ces mémoires contenait deux fils bien particuliers : les **RAS** et le **CAS**. Pour simplifier, le Signal RAS permettait sélectionner une ligne, et le signal CAS permettait de sélectionner une colonne.



Petite précision : les signaux RAS et CAS font quelque chose quand on les met à zéro et non à 1. Le mémoire va les prendre en compte quand on les fait passer de 1 à zéro : c'est à ce moment là que la ligne ou colonne dont l'adresse est sur le bus sera sélectionnée. Tant que des signaux sont à zéro, la ligne ou colonne reste sélectionnée : on peut changer l'adresse sur le bus, cela ne désélectionnera pas la ligne ou la colonne et la valeur présente lors du front descendant est conservée.

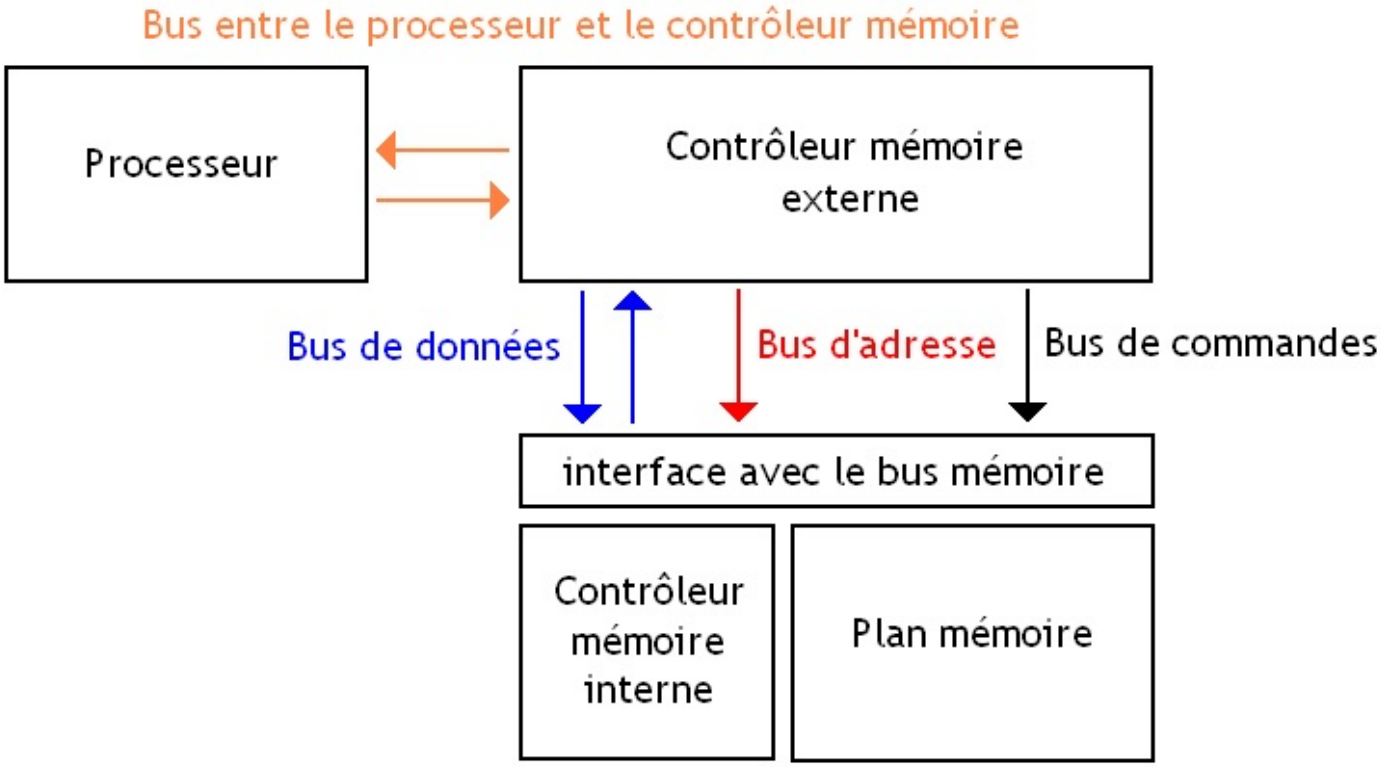
Dans la mémoire

Pour implémenter cette technique, nos mémoires FPM incorporaient deux mémoires tampons, deux registres, qui étaient chargés de stocker les numéros de colonnes et de ligne. Ceux-ci avaient leur sortie directement reliée aux décodeurs. A chaque signal RAS, le registre correspondant à une ligne était mis à jour avec le contenu du bus d'adresse. Idem pour le registre de colonne avec le CAS. On pouvait alors envoyer notre adresse en deux fois sans trop de dommages.



Contrôleur mémoire externe

Seul problème : notre processeur ne comprend que des adresses complètes : ces histoires d'adresses de lignes ou de colonnes, ça lui passe par dessus la tête. Le processeur envoie à la mémoire des adresses complètes. Pour communiquer avec la mémoire RAM, quelque chose doit découper ces adresses complètes en adresse de ligne et de colonne, et générer les signaux RAS et CAS. Ce quelque chose, c'est un circuit qu'on appelle **le contrôleur mémoire** externe.



Bien sûr, ce contrôleur mémoire là n'a rien à voir avec le contrôleur mémoire chargé de décoder les adresses vu dans les chapitre précédent qui est intégré dans la barrette de mémoire. C'est ainsi, on a deux contrôleurs mémoires : un placé sur la carte mère qui déduit quoi envoyer sur les bus en fonction de ce que demande le processeur, et un contrôleur intégré à nos barrette de mémoire qui décode les adresses et gère le sens de transfert.

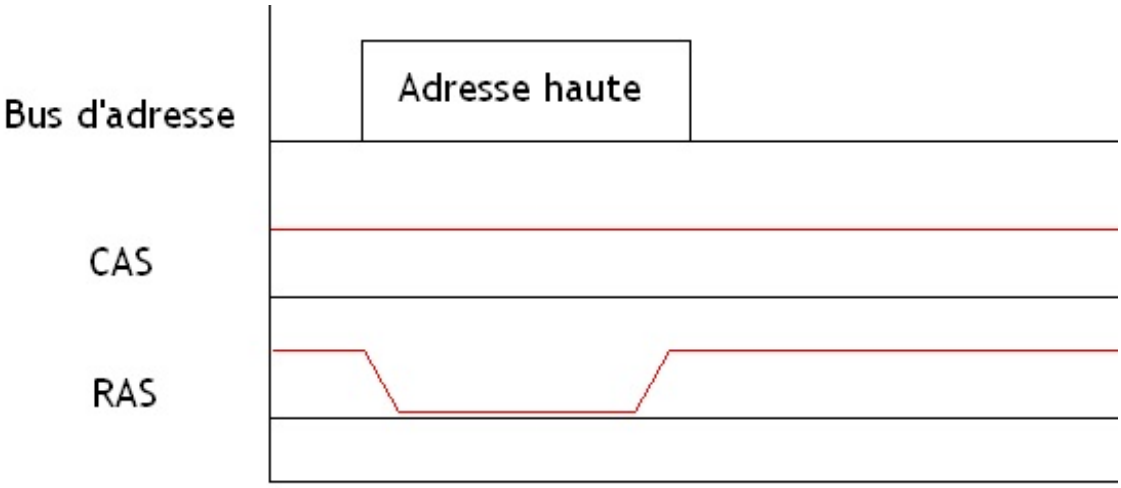
Rafraichissement mémoire

Ce contrôleur mémoire ne se charge pas que de la gestion des signaux CAS et RAS, ou du découpage des adresses. Il prend en charge pas mal d'autres fonctionnalités, et sert à beaucoup de choses. Autrefois, il s'occupait notamment du rafraichissement mémoire.

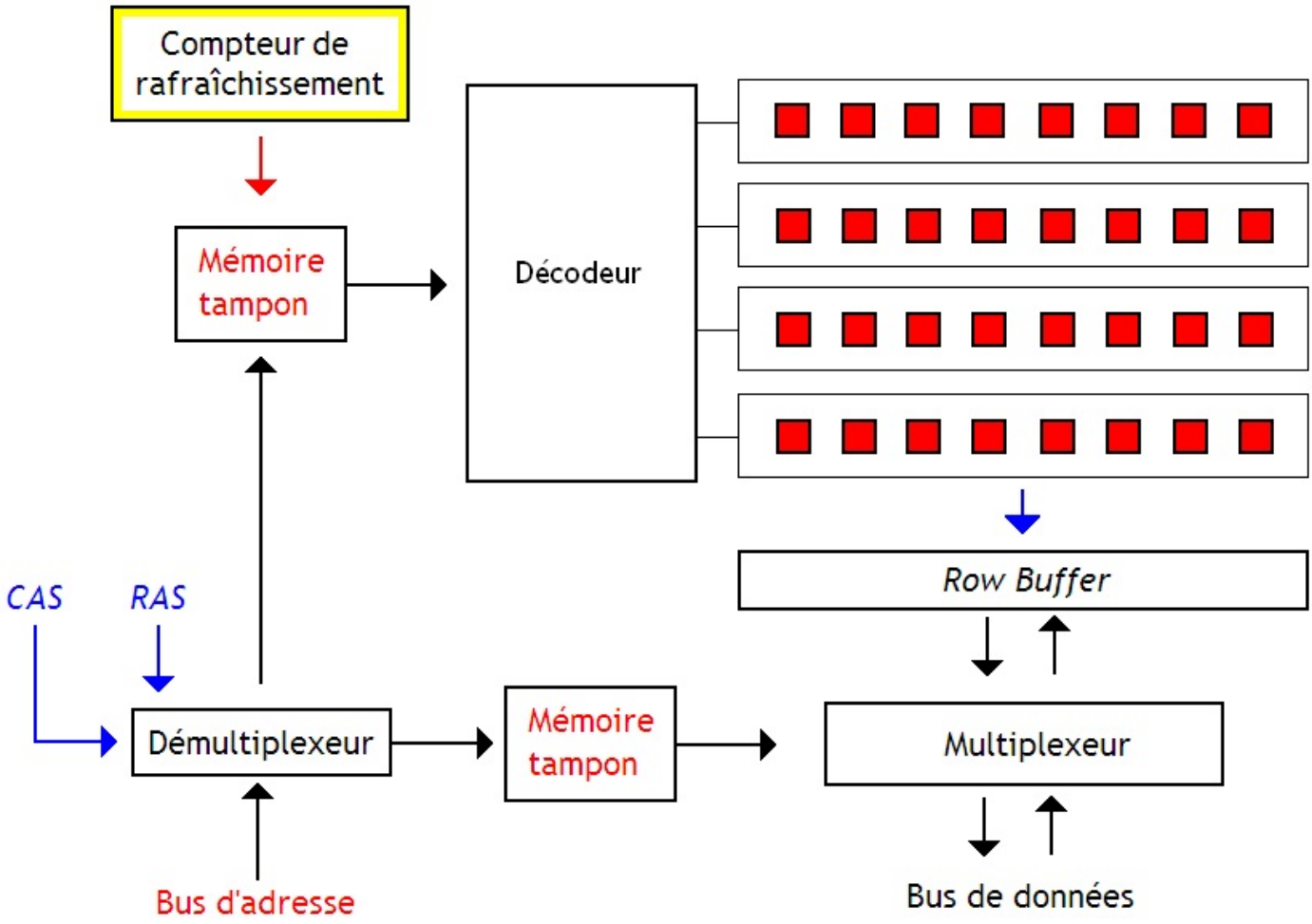
Il ne faut pas oublier quelque chose : ces mémoires FPM et EDO sont avant tout des mémoires DRAM, et doivent donc être rafraichies suffisamment souvent. Sur ces mémoires FPM et EDO, le rafraichissement se faisait ligne par ligne : on réécrivait chaque ligne une par une, à intervalle réguliers.

Pour donner l'ordre à la mémoire de rafraichir une ligne, il suffit de :

- placer l'adresse haute permettant de sélectionner la ligne rafraichir sur le bus d'adresse ;
- positionner le signal RAS à 0 ;
- et laisser CAS à 1.

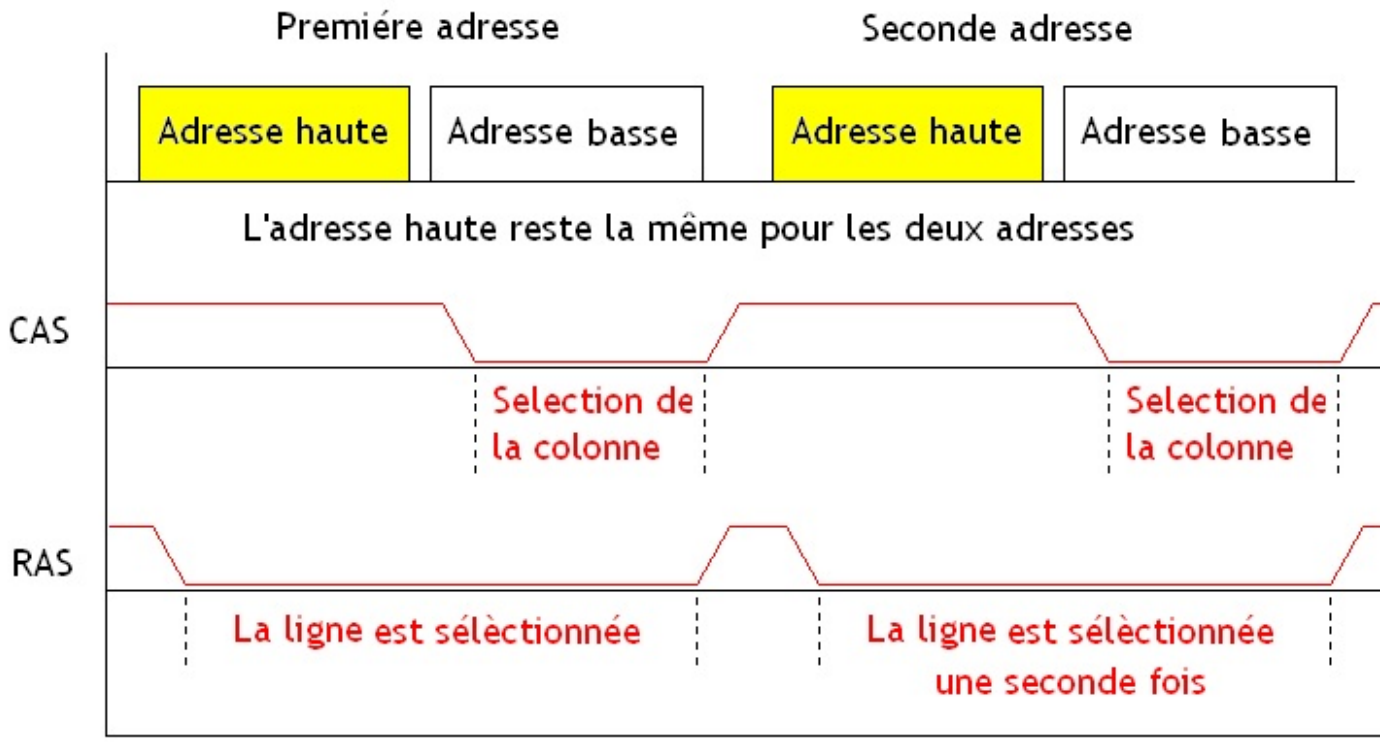


Rapidement, les constructeurs de mémoire se sont dit qu'il valait mieux gérer ce rafraichissement de façon automatique, sans faire intervenir le contrôleur mémoire intégré à la carte mère. Ce rafraichissement a alors été délégué au contrôleur mémoire intégrée à la barrette de mémoire, et est maintenant géré par des circuits spécialisés. Ce circuit de rafraichissement automatique n'est rien d'autre qu'un compteur, qui contient un numéro de ligne (celle à rafraichir).

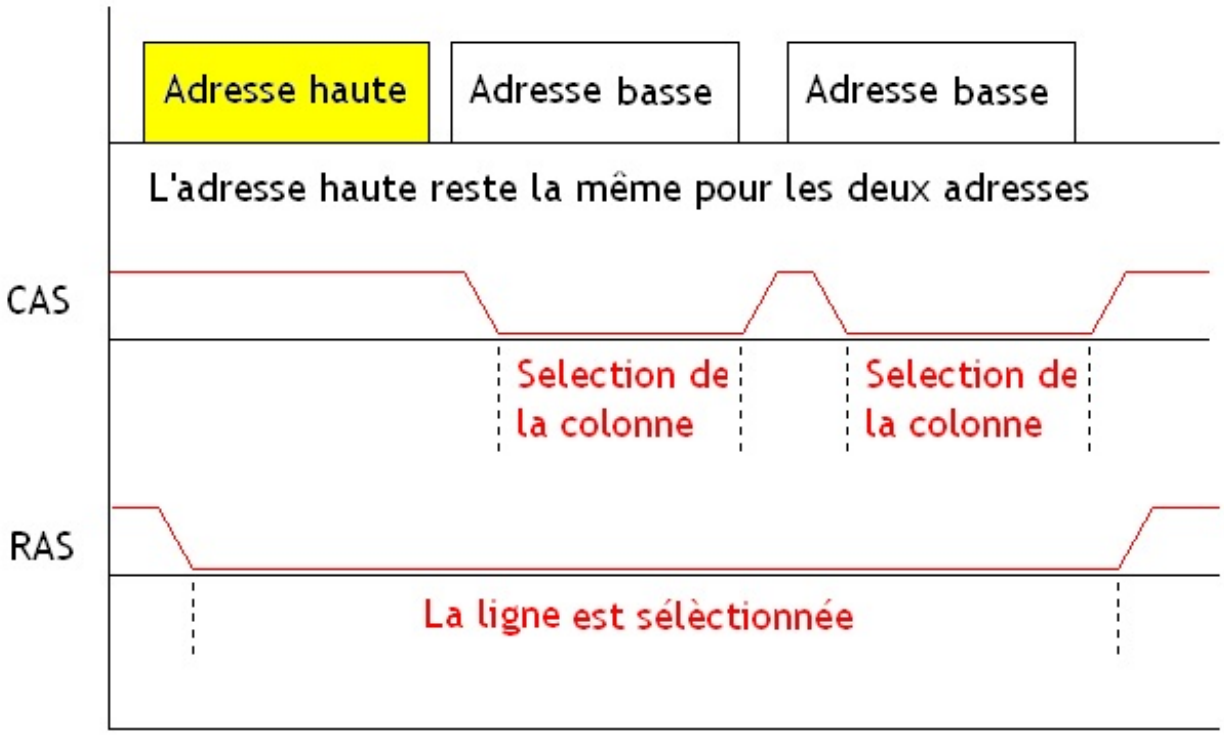


Mémoires FPM et EDO

Les mémoires FPM et EDO fonctionnaient de façon asynchrone, comme vu au dessus. Mais elles ont apportées une première amélioration comparée aux mémoires vues au chapitre précédent. Sur les anciennes mémoires (avant l'invention des mémoires FPM), à chaque fois qu'on voulait changer de case mémoire, on devait préciser à chaque fois la ligne et la colonne. Ainsi, si on voulait accéder à deux données placées dans des adresses proches et placées sur la même ligne, on devait sélectionner la même ligne deux fois : une par adresse.



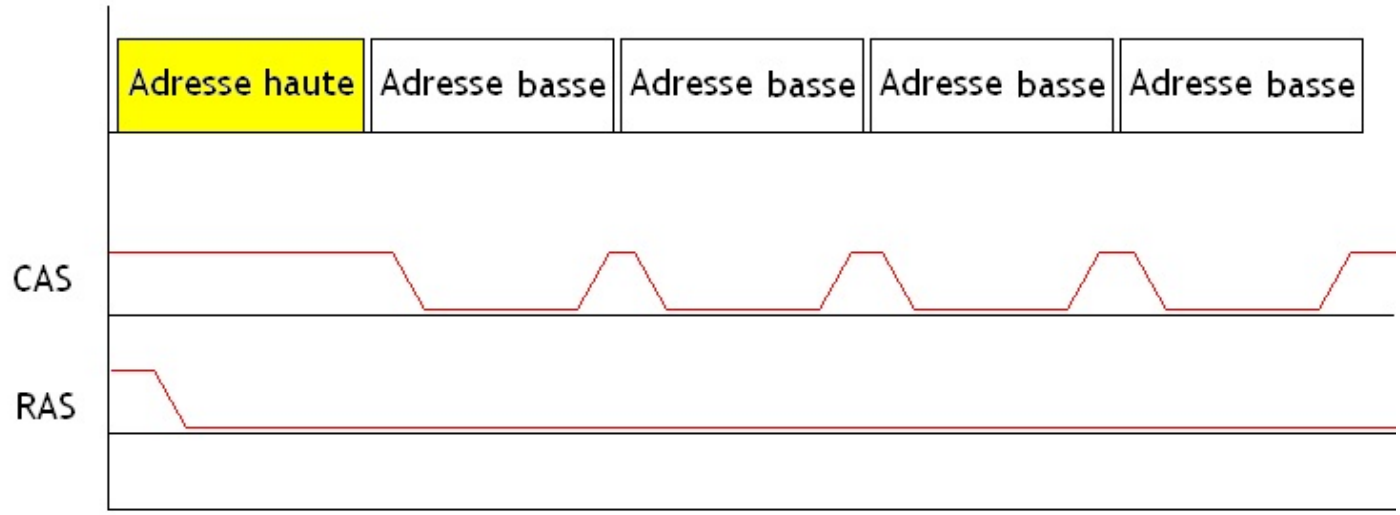
Avec la FPM ou l'EDO, on n'avait plus besoin de préciser deux fois la ligne si celle-ci ne changeait pas : on pouvait garder la ligne sélectionnée durant plusieurs accès.



EDO-RAM

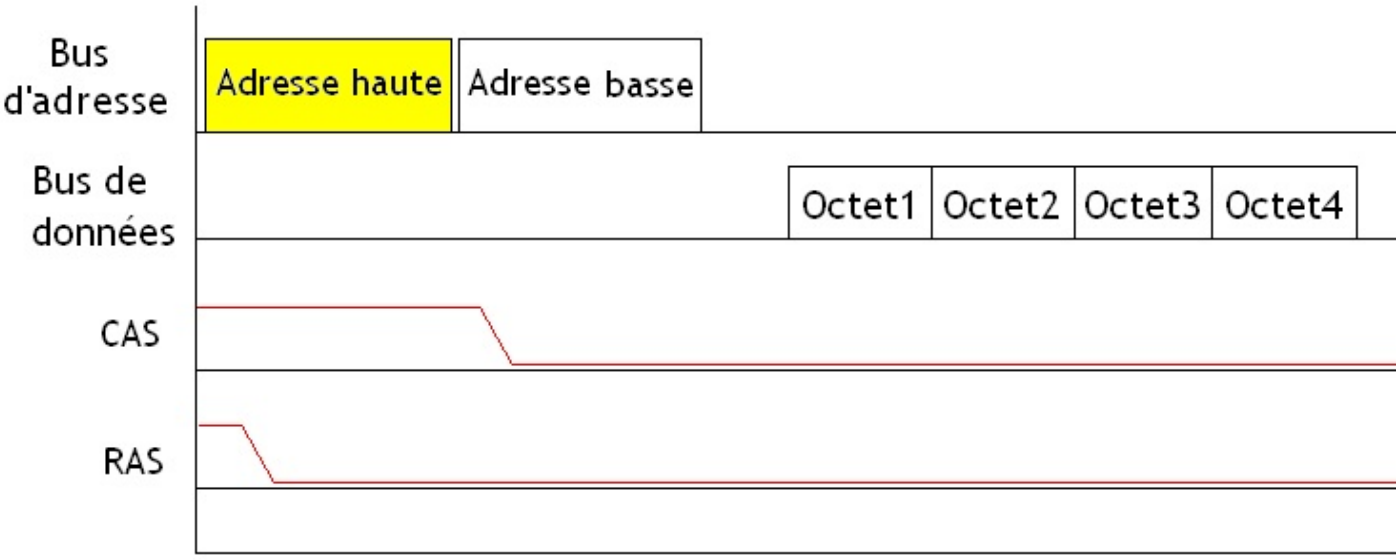
L'EDO-RAM a été inventée quelques années après la mémoire FPM. Il n'y a pas de grandes différences entre les mémoires EDO-RAM et les mémoires FPM. Cette mémoire EDO a étè déclinée en deux versions : la EDO simple, et la *Burst* EDO. L'EDO simple n'apportait que de faibles améliorations vraiment mineures, aussi je me permet de la passer sous silence. Mais pour la *Burst* EDO, c'est autre chose. Celle-ci permettait d'accéder à quatre octets consécutifs placés sur la même ligne bien plus rapidement que ses

prédécesseurs. En effet, sur les mémoires EDO et FPM, on devait lire ces 4 octets consécutifs colonnes par colonnes. Il fallait envoyer les adresses basses unes par unes (en réglant CAS comme il faut). Notre processeur devait donc envoyer l'adresse du premier octet, attendre que ligne et colonnes soient sélectionner, lire le premier octet, passer à la colonne suivante, lire le second octet, rechanger la colonne, etc. Chacune de ces étapes prenait un cycle d'horloge (on parle de l'horloge du processeur).

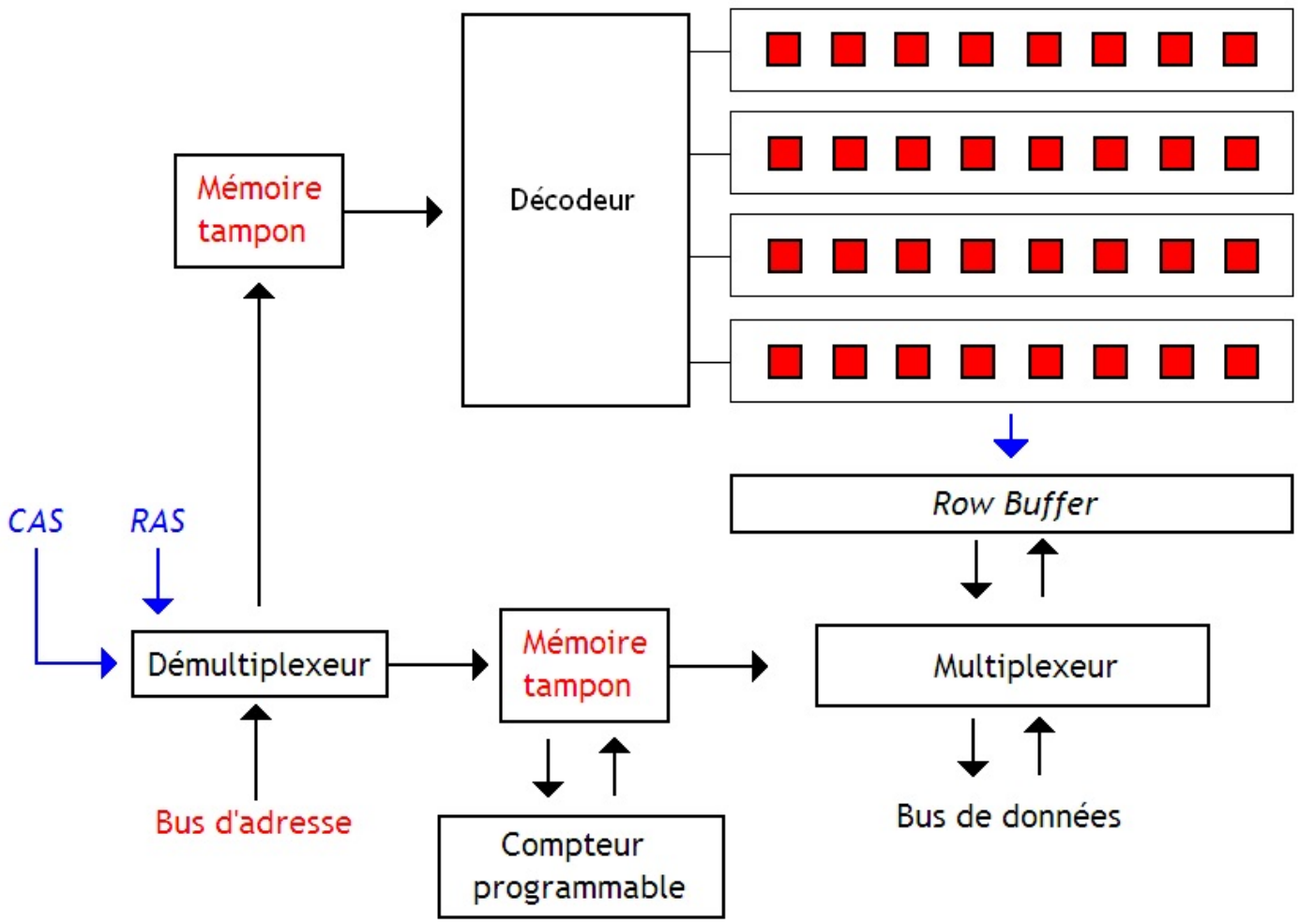


Avec les *Burst* EDO-RAM, on pouvait faire cela bien plus rapidement : pas besoin d'envoyer les adresses basses de chaque colonne unes par unes. On pouvait programmer notre mémoire pour que celle-ci effectue les 4 accès à ces 4 octets consécutifs toute seule, sans qu'on aie besoin de placer les adresses basses de chaque colonnes sur le bus et positionner le signal CAS : cela fait pas mal de temps de gagné.

Pour cela, il suffit de configurer le contrôleur mémoire pour lui ordonner d'effectuer un tel accès, et adresser le premier octet de la suite d'octets qu'on souhaite lire. Les 4 octets voulus étaient alors disponibles les uns après les autres : il suffisait d'attendre un cycle d'horloge par octet. Les cycles d'horloges nécessaires pour changer de colonne, nécessaires sur les mémoires FPM, étaient inutiles lorsqu'on accédait à des données successives. Ce genre d'accès mémoire s'appelle un accès en ***Burst*** ou en rafale.



Implémenter cette technique nécessite d'ajouter des circuits dans notre mémoire. Il faut notamment rajouter un compteur, capable de faire passer d'une colonne à une autre quand on lui demande. Le tout état accompagné de quelques circuits pour gérer le tout.



Les mémoires SDRAM

De nos jours, nous n'utilisons plus de mémoires FPM ou EDO : ces mémoires FPM et EDO ont laissées la place à des mémoires plus perfectionnées. Il faut dire que les mémoires FPM et EDO devinrent de plus en plus lentes au fil du temps et qu'il a fallu leur trouver un successeur. Leur successeur s'appelle la **mémoire SDRAM**. Cette mémoire n'est pas si différente des mémoires FPM ou EDO : on y retrouve toujours nos fameux signaux RAS et CAS, les accès en rafale, et les diverses améliorations apportées par les mémoires FPM et EDO. Mais il y a toute de même une grosse amélioration avec les mémoires SDRAM : ces mémoires ne sont pas asynchrones et sont synchronisées avec le bus par une horloge.

L'utilisation d'une horloge ne semble pas vraiment changer grand chose au premier abord. Mais en réalité, ça change beaucoup de choses. Premièrement, les temps mit pour lire ou écrire une donnée sont fixés et connus une fois pour toutes : le processeur sait qu'entre le moment où il déposera une adresse sur le bus d'adresse, et le moment où la mémoire aura terminé, il se passera un nombre fini (2, 3, 5, etc) de cycles d'horloge. Il peut donc déposer son adresse sur le bus, et faire ce qu'il veut dans son coin durant quelques cycles en attendant que la mémoire fasse ce qu'on lui demande.

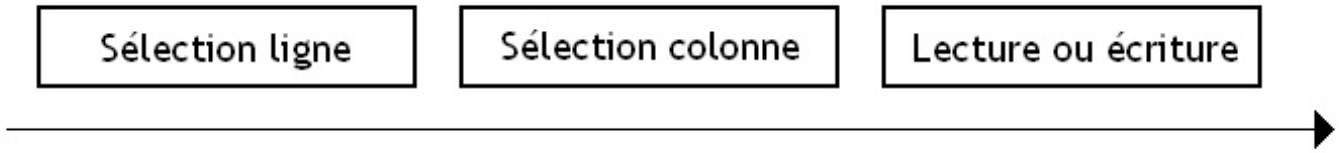
Avec les mémoires asynchrones, ce n'était pas possible : ces mémoires mettait un temps variable pour faire ce qu'on leur demandait. Le processeur ne faisait rien tant que la mémoire n'avait pas répondue : il exécutait ce qu'on appelle des ***wait state*** en attendant que la mémoire aie finie.

Pipelining des requêtes mémoires

Le fait que notre mémoire SDRAM soit reliée à une horloge (ainsi que quelques autres petites modifications) permet d'apporter une amélioration assez sympathique comparé aux mémoires EDO et FPM. Pour expliquer quelle est cette fameuses amélioration, quelques rappels s'imposent.

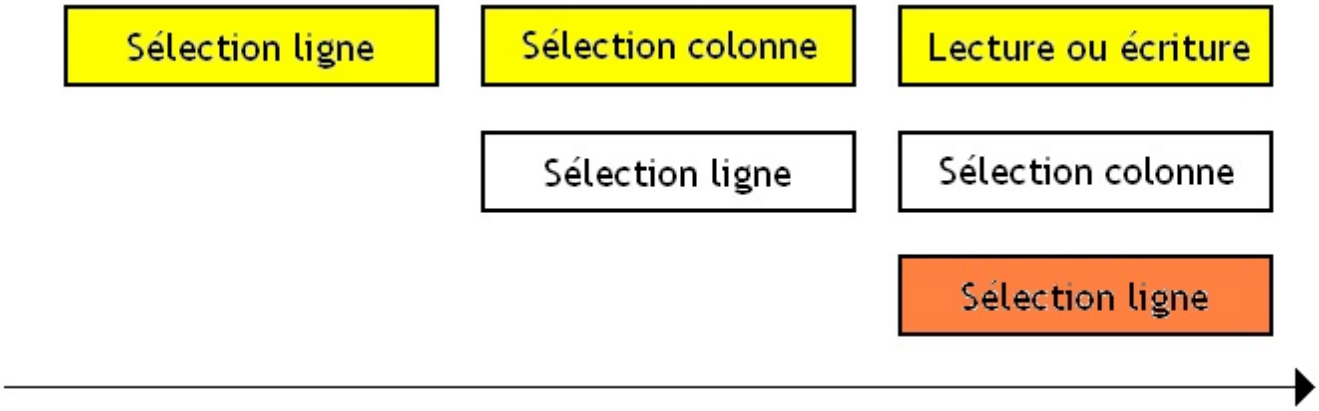
La sélection d'une case mémoire se fait en étapes : on commence par sélectionner la ligne, puis on sélectionne la colonne, et enfin on peut lire ou écrire notre donnée. Auparavant, on devait attendre qu'une lecture/écriture soit finie avant d'en envoyer une nouvelle, ce qui fait que ces étapes sont effectuées les unes après les autres.

On pouvait éventuellement passer outre certaines étapes inutiles : par exemple, il n'y avait pas besoin de sélectionner deux fois la même ligne depuis les mémoires FPM), mais on devait attendre d'avoir lu ou écrit notre donnée avant de demander à la mémoire d'accéder à la ligne/colonne suivante.



Avec les SDRAM, la situation est différente. Avec l'utilisation d'une horloge, ce n'est plus vraiment le cas : vu que la sélection d'une ligne ou d'une colonne peut prendre plusieurs cycles d'horloge, on peut envoyer une adresse complète avant même que la ligné ou la colonne soit sélectionnée. Cela permet d'envoyer à notre mémoire une demande de lecture ou d'écriture (en envoyant une adresse complète et autres bits de commande) sans attendre que les précédentes soient finies.

Pour faire simple, cela signifie qu'on est pas obligé de laisser les signaux RAS et CAS à zéro pendant qu'on sélectionne une ligne ou une colonne : il suffit de placer ceux-ci à zéro durant un cycle d'horloge pour la mémoire comprenne ce qu'on lui demande et laisse la place à une autre demande. Cela s'appelle faire du ***pipelining***.



Comme vous le voyez sur le schéma du dessus, une lecture/écriture, une sélection d'une colonne, ainsi qu'une sélection de ligne se font en même temps, mais dans des circuits différents. Cela est rendu possible en utilisant notre *Row Buffer* et en ajoutant quelques petits registres au bon endroit.

Néanmoins, il y a parfois des situations assez particulières pour lesquelles il n'est pas forcément possible d'effectuer des accès en mémoire ainsi. Pour être plus précis, sachez que la sélection d'une colonne ou d'une ligne peuvent prendre plusieurs cycles d'horloge et ne durent pas le même temps, ce qui complexifie la chose, sans compter qu'on peut parfois se retrouver avec des étapes supplémentaires.

Timings mémoires

Comme je l'ai brièvement mentionné plus haut, il faut un certain temps pour sélectionner une ligne ou une colonne. Mais dans notre mémoire, il existe d'autres temps de d'attente plus ou moins bien connus, qu'il est parfois important de connaitre. Dans cette partie, je vais vous lister quels sont ces temps de latence. Certains d'entre vous qui sont familiers avec l'*overcloking* connaissent ces temps d'attente sans le savoir : ces fameux temps d'attente, ou **timings mémoires** sont en effet très important pour eux. Aussi nous allons les voir en détail.

La façon de mesurer ces timings varie : sur les mémoires FPM et EDO, on les mesure en unités de temps (secondes, millisecondes, micro-secondes, etc), tandis qu'on les mesure en cycles d'horloge sur les mémoires SDRAM. Ainsi, si je vous dit qu'une mémoire DDR de marque xxxx et de numéro de série a un tRAS de 5, cela signifie qu'il faut attendre 5 cycles d'horloge avant que la ligne soit sélectionnée.

Timing	Description
tRAS	Le premier de ces timings s'appelle le **tRAS** : c'est le temps mit pour sélectionner une ligne.
tCAS	Le second timing s'appelle le **tCAS** et correspond au temps mit pour sélectionner une colonne. Comme le tRAS (et comme tous les autres timings), on le mesure en cycles d'horloge sur les SDRAM et les DDR. Il faut préciser une

tCAS	petite chose assez amusante : le CAS est un timings qui est programmable sur les toutes les mémoires SDRAM et DDR.
tRP	Nos mémoires RAM sont des mémoires à *Row Buffer*. Pour rappel, cela signifie que pour sélectionner une case mémoire à lire ou écrire, il faut sélectionner la ligne à lire et la recopier dans une mémoire tampon nommée le *Row Buffer*, et sélectionner la colonne. En fait, j'ai passé un détail sous silence : dans certains cas, il faut aussi penser à vider le *Row Buffer*. Lorsque l'on souhaite accéder à deux cases mémoires qui ne sont pas sur la même ligne, on doit vider le *Row Buffer*, qui contient encore la ligne précédente, avant de pouvoir sélectionner la ligne et la colonne. Le temps mit pour vider la ligne et la faire revenir à son état initial est appelé le ***tRP***.
tRCD	Vient ensuite le temps mit entre la fin de la sélection d'une ligne, et le moment où l'on peut commencer à sélectionner la colonne, qu'on appelle le **tRCD**.
tWTR	Une fois qu'on a écrit une donnée en mémoire, il faut un certain temps avant de pouvoir lancer une lecture qu'on appelle le **tWTR**.
tCAS-to-CAS	C'est le temps minimum entre deux sélections de deux colonnes différentes.

Rapidité

Ces timings influencent grandement la vitesse à laquelle on accède à une donnée dans la mémoire. Et oui, car suivant la disposition des données dans la mémoire, l'accès peut être plus ou moins rapide. Il existe ainsi quelques possibilités plus ou moins différentes, qu'on va vous citer.

Premier cas : la donnée que l'on cherche à lire est présente sur la même ligne que la donnée qui a été accédée avant elle. Cela se produit souvent lorsque l'on doit accéder à des données proches les unes des autres en mémoire. Dans ce cas, la ligne entière a été recopiée dans le *Row Buffer* et on n'a pas à la sélectionner : on doit juste changer de colonne. Ce genre de situation s'appelle un ***Row Buffer Hit***. Le temps nécessaire pour accéder à notre donnée est donc égal au temps nécessaire pour sélectionner une colonne (le tCAS auquel il faut ajouter le temps nécessaire entre deux sélections de deux colonnes différentes (le t-CAS-To-CAS).

Second cas : on accède à une donnée située dans une ligne différente : c'est un ***Row Buffer Miss***. Et là, c'est une catastrophe ! Dans ce genre de cas, il faut en effet vider le *Row Buffer*, qui contient la ligne précédente, en plus de sélectionner la ligne et la colonne. On doit donc ajouter le tRP au tRAS et au tCAS pour avoir le temps d'accès total à notre donnée.

Le SPD

Évidemment, ces timings ne sont pas les mêmes suivant la barrette de mémoire que vous achetez. Certaines mémoires sont ainsi conçues pour avoir des timings assez bas et sont donc plus rapides, et surtout : beaucoup plus chères que les autres. Le gain en performances dépend beaucoup du processeur utilisé et est assez minime comparé au prix de ces barrettes. Les circuits de notre ordinateur chargés de communiquer avec la mémoire (ceux placés soit sur la carte mère, soit dans le processeur), doivent connaitre ces timings et ne pas se tromper : sans ça, l'ordinateur ne fonctionne pas.

Pour cela, notre barrette de mémoire contient une petite mémoire ROM qui stocke les différents timings d'une façon bien déterminée : cette mémoire s'appelle le ***Serial Presence Detect***, aussi communément appelé le **SPD**. Ce SPD contient non seulement les timings de la mémoire RAM, mais aussi diverses informations, comme le numéro de série de la barrette, sa marque, et diverses informations.

Le contenu de ce fameux SPD est standardisé par un organisme nommé le JEDEC, qui s'est chargé de standardiser le contenu de cette mémoire, ainsi que les fréquences, timings, tensions et autres paramètres des mémoires SDRAM et DDR. Cette mémoire ROM est lue au démarrage de l'ordinateur par certains circuits de notre ordinateur (le fameux BIOS, allez voir ici, pour les curieux : le BIOS, qu'est-ce que c'est ?), afin de pourvoir configurer ce qu'il faut.

Mode Burst

Les mémoires SDRAM possèdent aussi un mode *Burst*, qui a toutefois été amélioré comparé au *Burst* des mémoires EDO : il est devenu **programmable** ! Pour cela, notre mémoire RAM contient un petit registre, le ***register mode***, qui permet de configurer notre mémoire RAM et plus précisément le fonctionnement de son mode *Burst*. Il existe ainsi un bit qui permettra de préciser si on veut effectuer des accès normaux (le bit est alors mit à 1), ou des accès en *Burst* (le bit est mit à zéro).

Avec les SDRAM, on peut spécifier le nombre d'octets consécutifs auxquels on veut accéder. Sur les mémoires EDO, on devait

absolument lire 4 octets uns par uns, sans pouvoir faire plus ou moins, mais les mémoires SDRAM corrigent ce problème. D'autres bits vont ainsi permettre de configurer le nombre de cases mémoires consécutives auquel on doit accéder lors d'un accès en *Burst*. On peut ainsi accéder à 1, 2, 4, ou 8 octets en une seule fois.

Qui plus est, il existe deux types d'accès en *Burst* sur les SDRAM : l'accès *interleaved*, let l'accès séquentiel. le mode séquentiel est le mode Burst normal : on accède à des octets consécutifs les uns après les autres. Un bit du *register mode* permet de sélectionner le type de *burst* voulu. Le mode *interleaved* le fait dans un ordre légèrement différent qu'on ne va pas voir ici (celui-ci n'apporte rien de vraiment utile).

Burst ordering

Il faut noter que ces accès en *Burst* doivent répondre à certaines contraintes : cela ne marche correctement que dans des blocs dont la taille est celle du bus de donnée et qui sont placés à des adresses bien précises.

Dans le cas contraire, il y a quelques petites subtilités qui font leur apparition. En fait, tout se passe comme si notre ligne était découpée en blocs ayant la même taille que le bus de données, ce qui donne des blocs de 8 cases mémoires. Dans ce qui va suivre, on va noter les cases mémoires appartenant à un de ces blocs 1, 2, 3, 4, 5, 6, 7, 8 et 9. Un accès en mode *Burst* n'est pas obligé de commencer par lire ou écrire le bloc 1 : on peut très bien commencer par lire ou écrire au bloc 3, par exemple.

Mais que ce passe-il lorsque l'on veut effectuer un accès en *Burst* sur 8 cases mémoires ?

Et bien regardons ce qui se passe. On commence par accéder à la case mémoire numérotée 3, puis la 4, la 5 , la 6 et la 7. Il reste encore 3 cases mémoires à lire, mais on arrive au bout de notre bloc de 8 cases mémoires. On pourrait imaginer divers scénarios : passer au bloc suivant semble être le plus logique. Mais ce n'est pas ce qui se passe : l'accès reprend au bloc 1, et on accède aux blocs 1, 2 et 3. En clair : une fois arrivé au bout de notre bloc de 8 cases mémoires, on reprend au début. Avouez que vous êtes surpris.

Les mémoires DDR

Nos processeurs sont de plus en plus exigeants, et la vitesse de la mémoire est un sujet primordial. Pour augmenter la vitesse de la mémoire, la solution la plus évidente est d'augmenter sa fréquence. Mais le seul problème, c'est qu'augmenter une fréquence ne se décrète pas comme ça ! Il faut dire que le plan mémoire ne peut pas vraiment être rendu plus rapide, pour des tas de raisons techniques difficiles à comprendre. Augmenter la fréquence du plan mémoire n'est donc pas une solution.

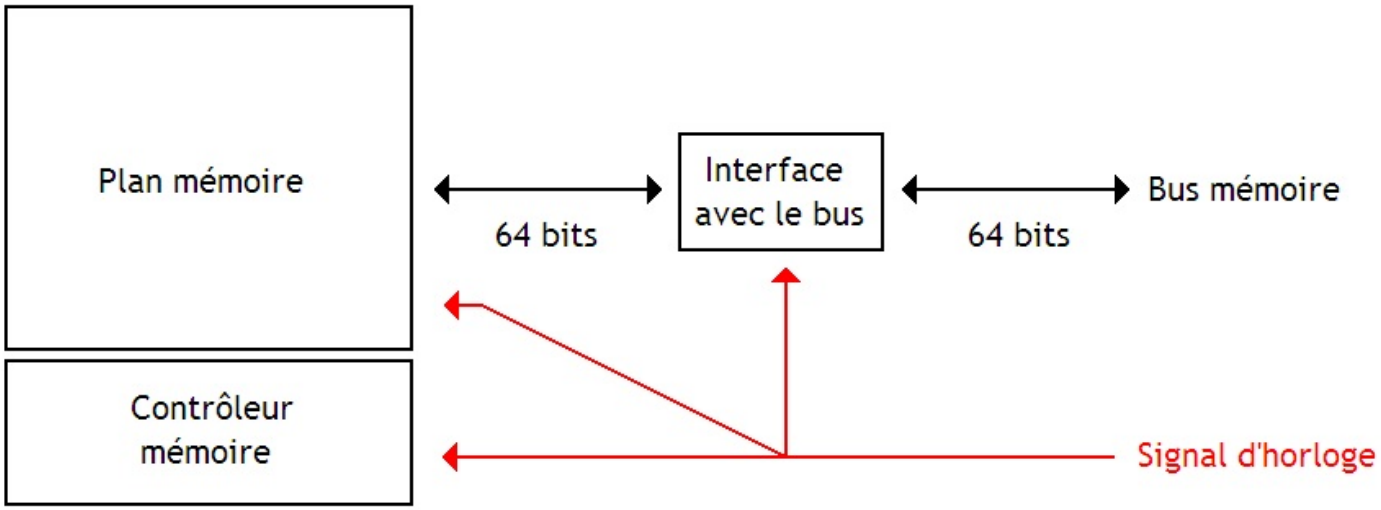
Une autre solution pourrait être d'augmenter le débit de la mémoire : on n'augmente pas sa fréquence, mais on lui permet de charger plus de données en une seule fois. Ainsi, au lieu d'aller lire ou écrire 64 bits d'un coup (avec l'accès en rafale, ou par d'autres techniques), on peut lui permettre de lire ou d'écrire plus de données d'un coup. Cette solution serait une bonne solution : les programmes ordinaires ont souvent besoin d'accéder à des données consécutives, sans compter la présence de caches qui peut encore exacerber ce phénomène. Charger plus de données consécutives d'un coup est souvent une bonne idée.

Seul problème : il faudrait rajouter des broches sur la mémoire et câbler plus de fils pour faire transiter ces bits supplémentaires. Le prix de la mémoire s'envolerait, et elle serait bien plus difficile à concevoir. Sans compter les difficultés pour faire fonctionner l'ensemble à haute fréquence. Mais ce n'est pas pour autant qu'on va retourner à la case départ.

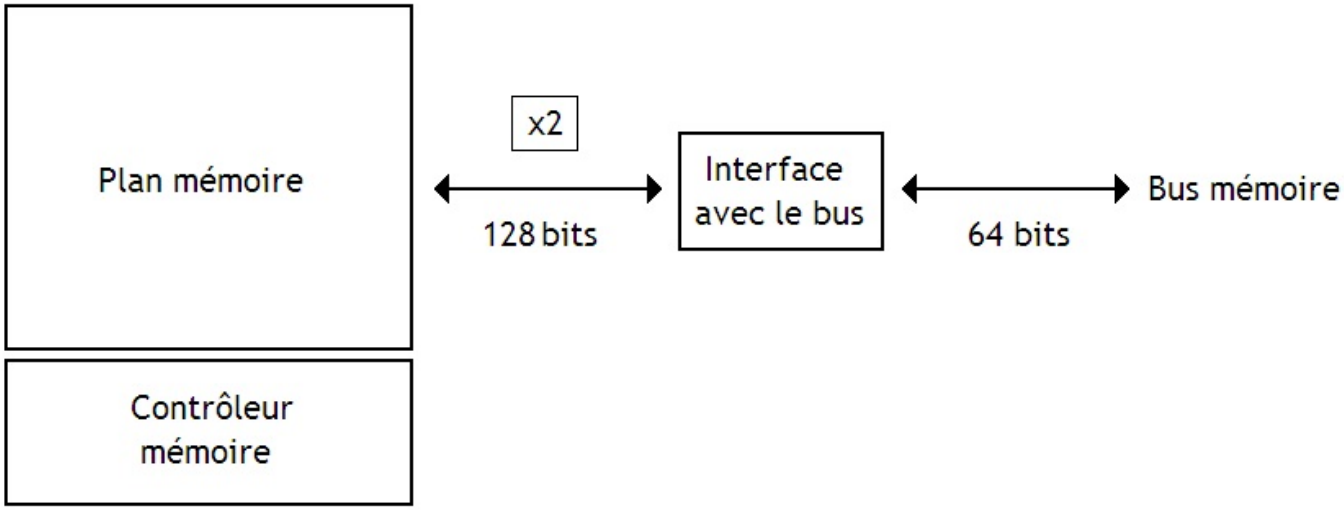
Il existe une solution un peu alternative, qui est une sorte de mélange des deux techniques. Ce compromis a donné naissance aux **mémoires DDR**. Il s'agit de mémoires SDRAM améliorées, avec une interface avec la mémoire légèrement bidouillée.

Principe

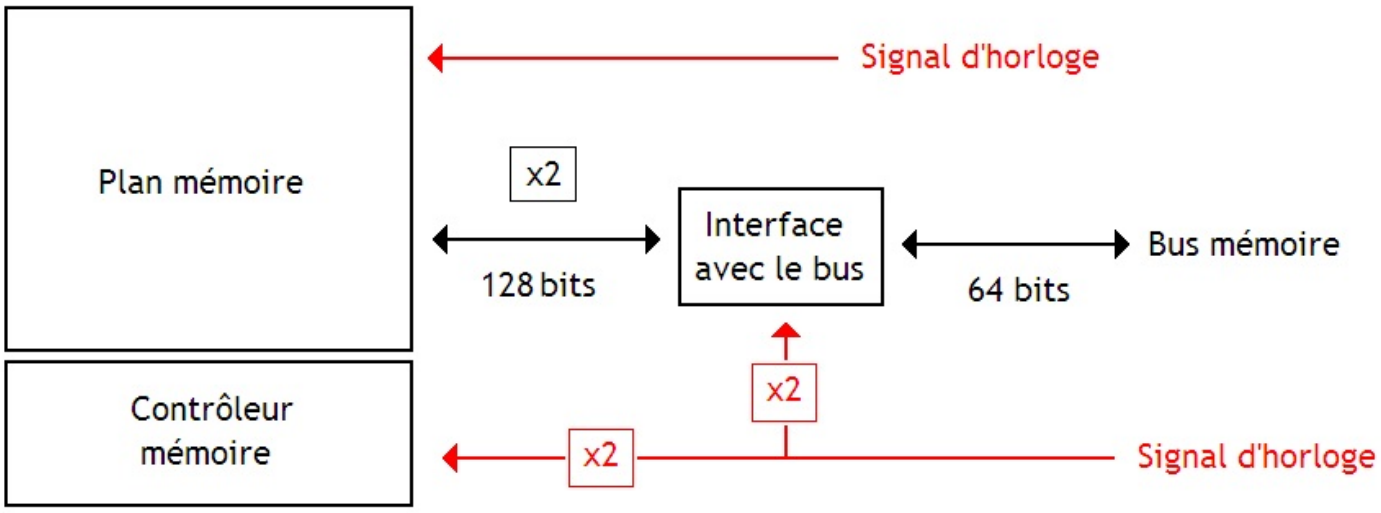
Dans nos mémoires SDRAM, les opérations internes à la mémoire sont synchronisées entre elles par une horloge. Sur les mémoires SDRAM simples, cette horloge est la même que celle du bus mémoire : mémoire et bus sont synchronisées de façon identique. A chaque cycle d'horloge, une SDRAM peut envoyer une seule donnée sur le bus. De plus, la taille du bus mémoire est identique au nombre de donnée pouvant être lues ou écrites dans la mémoire. En clair : le bus mémoire fait 64 bits, et la mémoire est capable de lire ou d'écrire dans 64 bits d'un coup. On dit merci au mode *Burst*.



Mais avec les mémoires DDR, tout change. Comme je l'ai dit, il s'agit d'une solution hybride. Le plan mémoire fonctionne toujours à la même fréquence : vu qu'on ne peut pas le rendre plus rapide, il fonctionne toujours à la même vitesse. Par contre, ce plan mémoire est modifié de façon à être plus large : on peut y lire ou y écrire 2, 4, 8 fois plus de données d'un seul coup. Par contre, le bus ne change pas ! Comme je l'ai dit, rajouter des fils et des broches n'est pas gratuit et pose beaucoup de problèmes. Donc, le bus transfère toujours autant de données en une seule fois.



Logiquement, vous devriez être étonnées : on charge plus de données depuis la mémoire que le bus ne semble supporter. Mais il y a un truc : la fréquence du bus est doublée. En gros, notre mémoire va lire 128 bits de données depuis le plan mémoire, et va les envoyer sur le bus par blocs de 64 bits.

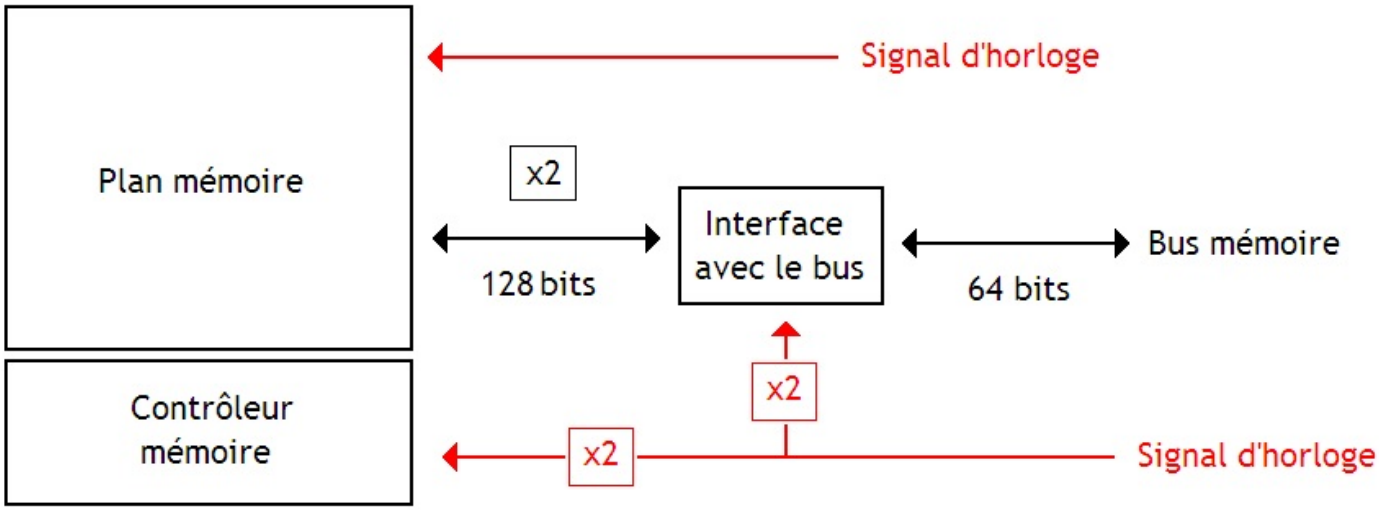


C'est ce qui différencie les SDRAM des mémoires DDR : leur contrôleur mémoire et le bus mémoire fonctionne à une fréquence qui est un multiple du plan mémoire. En contrepartie, le débit du plan mémoire est plus grand pour compenser. Ceci dit, cette organisation ne se fait pas sans modifications : dans l'exemple au dessus, il faut bien trouver un moyen pour découper notre bloc de 128 bits en deux blocs de 64, à envoyer sur le bus dans le bon ordre. Cela se fait dans l'interface avec le bus, grâce à une sorte de mémoire tampon un peu spéciale, dans laquelle on accumule les 128 bits lus ou à écrire.

D'autres différences mineures existent entre les SDRAM et les mémoires DDR. Par exemple, la tension d'alimentation des mémoires DDR est plus faible que pour les SDRAM. Ces mémoires DDR ont étés déclinées en plusieurs versions : DDR1, DDR2, DDR3, etc. Les différences tiennent dans la tension d'alimentation, leur fréquence, etc. Ces mémoires sont standardisées, et seuls certaines fréquences sont autorisées et agrées. L'organisme chargé de spécifier et de standardiser les mémoires s'appelle le JEDEC : c'est un consortium dont le rôle est des standardiser certains composants électroniques mis sur le marché.

DDR1

Dans le principe, les mémoires DDR1 transfèrent des données sur le bus à une fréquence deux fois supérieure à la fréquence du plan mémoire. En conséquence, leur débit parait doublé comparé à une mémoire SDRAM de même fréquence : elles peuvent transmettre deux fois plus de données dans des conditions favorables.



Mais dans les faits, seul un signal d'horloge est utilisé, que ce soit pour le bus, le plan mémoire, ou le contrôleur. Seulement, le bus et les contrôleur mémoire réagissent à la fois sur les fronts montants et sur les fronts descendants de l'horloge. Le plan mémoire, lui, ne réagit qu'aux fronts montants.

La quantité maximale de donnée qui peut être transmise par seconde par notre mémoire s'appelle son **débit théorique maximal**. Sur les mémoires SDRAM, ce débit théorique maximal se calculait en multipliant la largeur du bus de données (le nombre de bits qu'il peut transmettre en une fois) par sa fréquence. Par exemple, une mémoire SDRAM fonctionnant à 133 Mhz, et utilisée en simple channel utilisera un bus de 8 octets, ce qui fera un débit de $8 \times 133 \times 1024 \times 1024$ octets par seconde, ce qui fait environ du 1 giga-octets par secondes.

Pour les mémoires DDR1, il faut multiplier la largeur du bus mémoire par la fréquence, et multiplier le tout par deux pour obtenir le débit maximal théorique. En reprenant notre exemple d'une mémoire DDR fonctionnant à 200 Mhz, et utilisée en simple channel utilisera un bus de 8 octets, ce qui donnera un débit de $8 \times 200 \times 1024 \times 1024$ octets par seconde, ce qui fait environ du 2.1 giga-octets par secondes.

Ça peut sembler beaucoup, et c'est normal : c'est beaucoup ! Mais ce qui compte dans les performances d'un ordinateur, c'est surtout le temps d'accès. Certains programmes sont en effet très sensibles au temps mit pour accéder à notre mémoire : les jeux vidéos, par exemple, on besoin d'accéder rapidement à la mémoire. Une mémoire ayant un temps d'accès faible permettra d'éviter au processeur d'attendre les données qu'il doit manipuler. Il existe bien certains programmes qui ont besoin d'accéder à de grosses données, et pour lesquels avoir un débit élevé est important, mais ceux-ci ne sont pas vraiment une majorité.

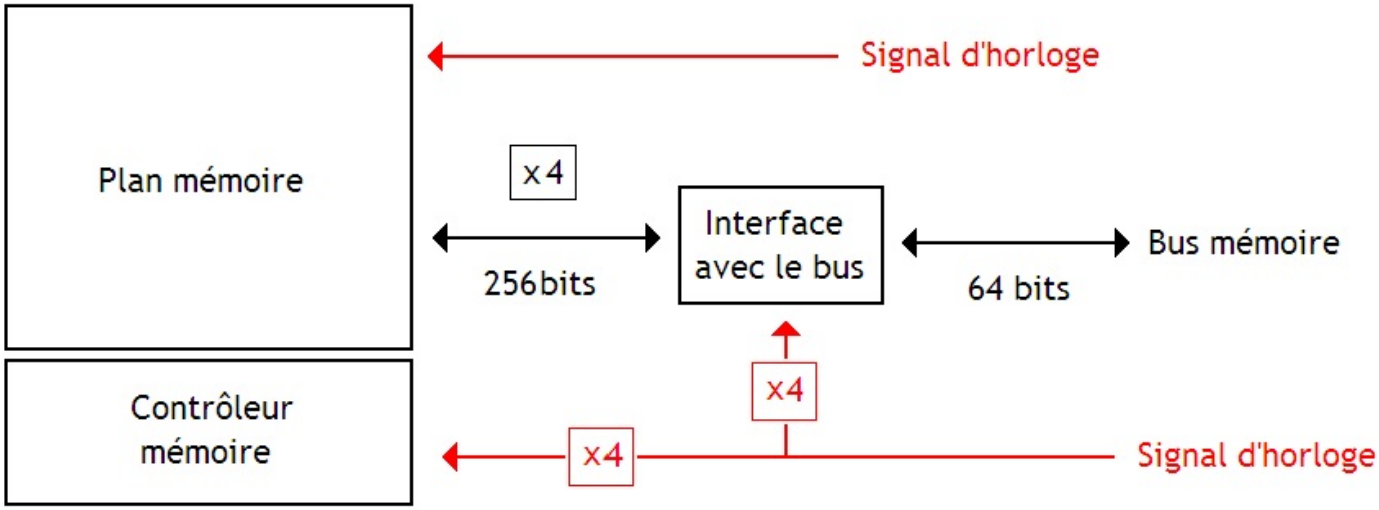
Il existe quatre types de mémoires DDR officialisés par le JEDEC.

Nom standard	Nom des modules	Fréquence du bus	Débit	Tension d'alimentation

DDR 200	PC-1600	100 Mhz	1,6 gibioctets seconde	2,5 Volts
DDR 266	PC-2100	133 Mhz	2,1 gibioctets seconde	2,5 Volts
DDR 333	PC-2700	166 Mhz	2,7 gibioctets seconde	2,5 Volts
DDR 400	PC-3200	200 Mhz	3,2 gibioctets seconde	2,6 Volts

DDR2

Dans le principe, es mémoires DDR2 transfèrent des données sur le bus à une fréquence quatre fois supérieure à la fréquence du plan mémoire. En conséquence, leur débit parait quadruplé comparé à une SDRAM de même fréquence. Pour obtenir leur débit théorique maximal, il faut multiplier la largeur du bus mémoire par la fréquence, et multiplier le tout par quatre pour obtenir le débit maximal théorique.



Mais dans les faits, le bus a une fréquence 2 fois supérieure à la fréquence de la mémoire et les transferts se font sur les fronts montants et les fronts descendants.

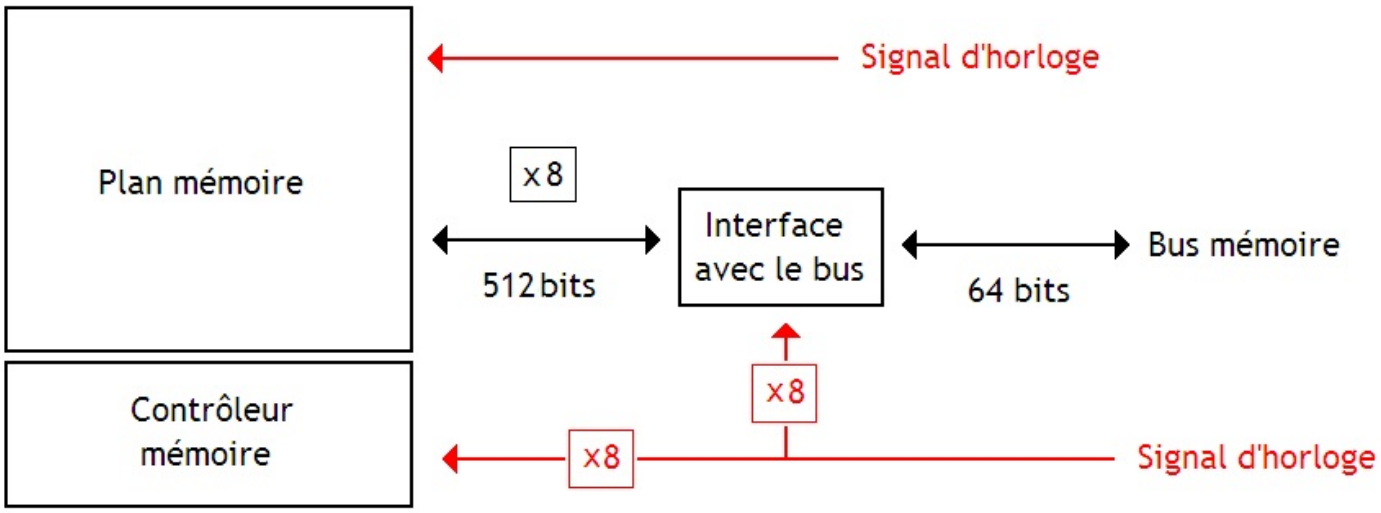
Avec les mémoires DDR2, 5 types de mémoires sont officialisées par le JEDEC.

Nom standard	Nom des modules	Fréquence du bus	Débit
DDR2 400	PC2-3200	100 Mhz	3,2 gibioctets par seconde
DDR2 533	PC2-4200	133 Mhz	4,2 gibioctets par seconde
DDR2 667	PC2-5300	166 Mhz	5,3 gibioctets par seconde
DDR2 800	PC2-6400	200 Mhz	6,4 gibioctets par seconde
DDR2 1066	PC2-8500	266 Mhz	8,5 gibioctets par seconde

Diverses améliorations ont étés apportées sur les mémoires DDR2 : la tension d'alimentation est notamment passée de 2,5/2,6 Volts à 1,8 Volts.

DDR3

Dans le principe, es mémoires DDR3 transfèrent des données sur le bus à une fréquence huit fois supérieure à la fréquence du plan mémoire. En conséquence, leur débit parait 8 fois supérieur comparé à une SDRAM de même fréquence. Pour obtenir leur débit théorique maximal, il faut multiplier la largeur du bus mémoire par la fréquence, et multiplier le tout par 8 pour obtenir le débit maximal théorique.



Mais dans les faits, le bus a une fréquence 4 fois supérieure à la fréquence de la mémoire et les transferts se font sur les fronts montants et les fronts descendants.

Avec les mémoires DDR3, 6 types de mémoires sont officialisées par le JEDEC.

Nom standard	Nom des modules	Fréquence du bus	Débit
DDR3 800	PC2-6400	100 Mhz	6,4 gibioctets par seconde
DDR3 1066	PC2-8500	133 Mhz	8,5 gibioctets par seconde
DDR3 1333	PC2-10600	166 Mhz	10,6 gibioctets par seconde
DDR3 1600	PC2-12800	200 Mhz	12,8 gibioctets par seconde
DDR3 1866	PC2-14900	233 Mhz	14,9 gibioctets par seconde
DDR3 2133	PC2-17000	266 Mhz	17 gibioctets par seconde

Diverses améliorations ont étés apportées sur les mémoires DDR3 : la tension d'alimentation est notamment passée à 1,5 Volts.

GDDR

Il existe enfin d'autres types de mémoires DDR : les **mémoires GDDR**, utilisées presque exclusivement sur les cartes graphiques. Contrairement aux autres, celles-ci ne sont pas vraiment standardisées par le JEDEC. Il en existe plusieurs types pendant que j'écris ce tutoriel : GDDR, GDDR2, GDDR3, GDDR4, et GDDR5. Mais attention : une mémoire GDDR2 n'a pas grand chose à voir avec une mémoire DDR2, par exemple. Il y a des différences (sauf pour la GDDR3 qui est identique à de la DDR3, mais c'est une exception) et il ne faut pas laisser piéger par les noms de ces mémoires, qui ressemblent à leur congénères crées sous la forme de barrettes.

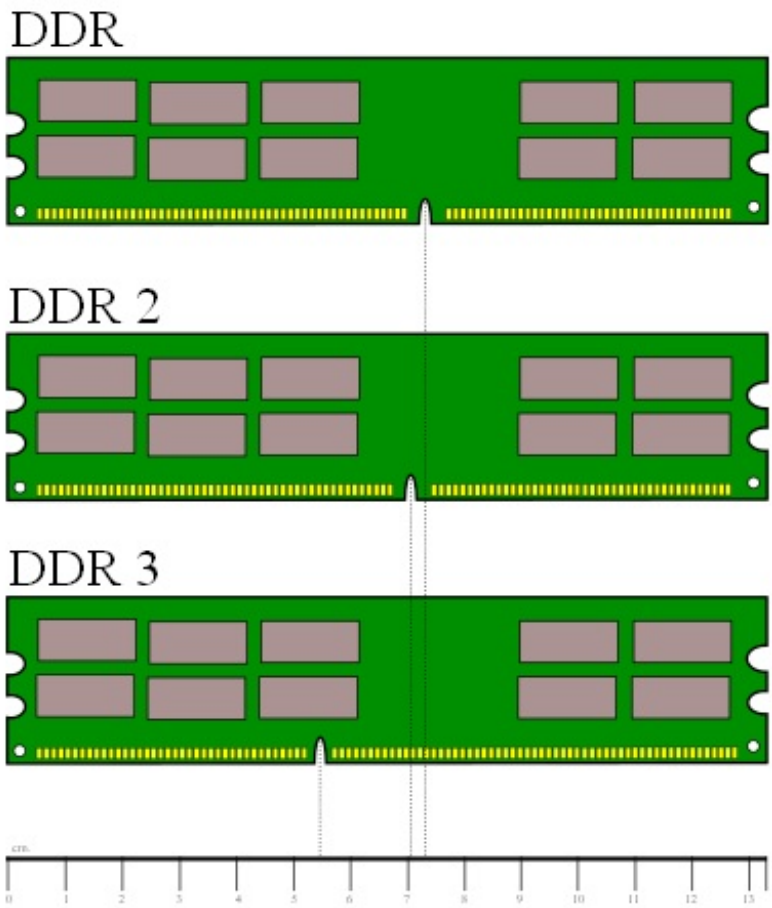
Généralement, les mémoires GDDR ont une fréquence plus élevée que leur congénères utilisées en tant que mémoire principale. Par contre, certains de leurs temps d'accès sont beaucoup plus élevés et peuvent aller jusqu'à 10 cycles d'horloge : sélectionner une ligne prend du temps. L'exception vient du temps de sélection d'une colonne, qui est assez faible : 1 cycle d'horloge, guère plus.

Cela permet ainsi à nos cartes graphiques d'accéder rapidement à des données consécutives en mémoire, tandis que les autres types d'accès sont vraiment lents. Qui plus, les mémoires GDDR sont souvent des mémoires multiports, ce qui permet d'accéder à plusieurs cases mémoires (pas forcément consécutives) en lecture ou en écriture en une seule fois.

Format DIMM et SO-DIMM

Les barrettes de mémoires SDRAM et DDR sont différentes des barrettes de mémoire FPM et EDO. Le format de celles-ci varie

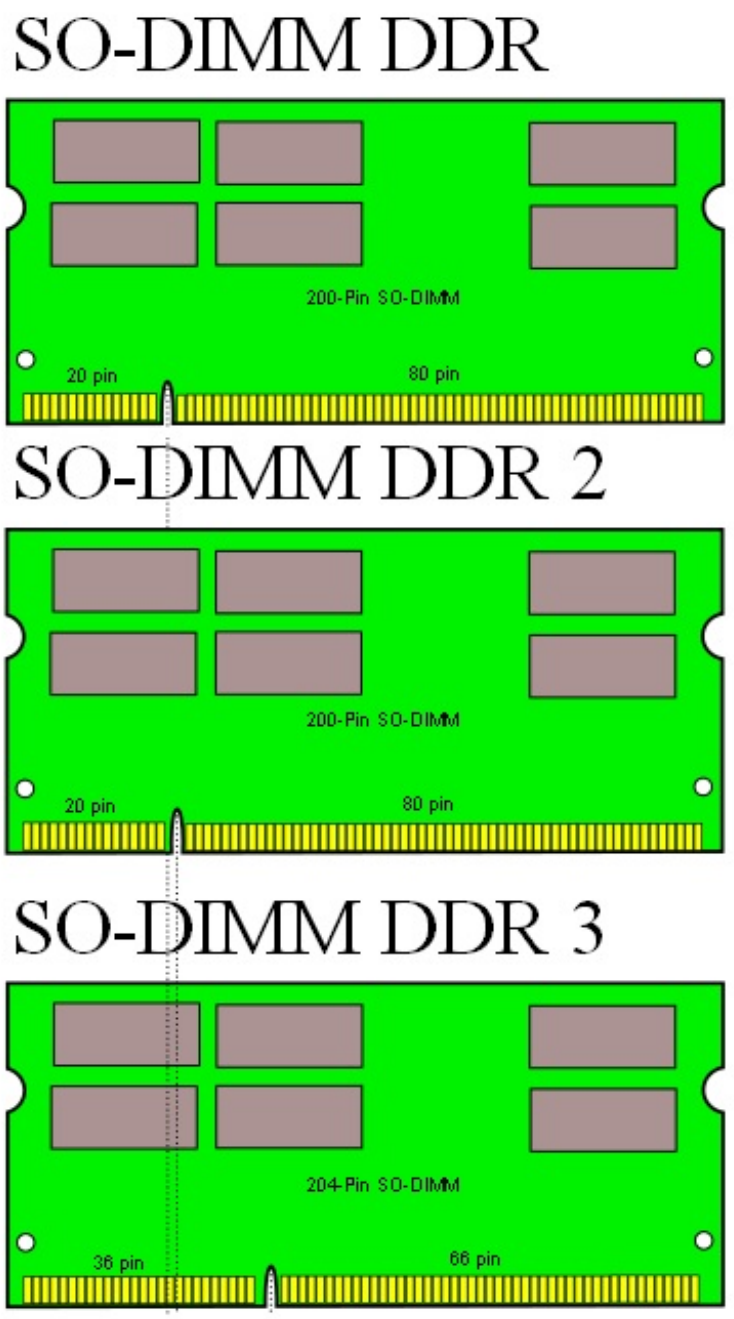
suivant la barrette, ainsi que le nombre de broches utilisées. Généralement, les barrettes utilisées sur les PC de bureau sont des barrettes au format **DIMM** : les deux cotés de la barrette sont utilisés pour placer les broches différentes, ce qui permet d'en mettre deux fois plus.



Le nombre de broches d'une barrette au format DIMM peut varier suivant la barrette utilisée, ainsi que le type de mémoire. On peut ainsi avoir entre 168 et 244 broches sur une seule barrette. Je suppose que vous comprendrez le fait que je ne souhaite pas vraiment en faire la liste, comme je l'ai pour les mémoires FPM 30 broches. Mais je vais quand même vous donner le nombre de broches par barrette en fonction du type de mémoire.

Mémoire	SDRAM	DDR1	DDR2	DDR3
Nombre de broches	168	184	214, 240 ou 244 suivant la barrette ou la carte mère	204 ou 240 suivant la barrette ou la carte mère

Les barrettes de mémoire des ordinateurs portables utilisent souvent un autre format de barrette : le SO-DIMM, et parfois un de ses concurrent : le Micro-DIMM.



Mémoire	SDRAM	DDR1 et DDR2	DDR3
Nombre de broches	144	200	204

Mémoires non-volatiles

Dans les chapitres précédents, nous avons vu comment fonctionnaient les mémoires RAM et nous les avons vues en détail. C'est maintenant au tour des mémoires de masse et des mémoires de stockage d'avoir leur quart d'heure de gloire. Dans ce chapitre, nous allons aborder les mémoires FLASH et le disque dur.

Le disque dur

Et maintenant, j'ai le plaisir de vous annoncer que nous allons étudier la plus célèbre des mémoires de masse : j'ai nommé, le **disque dur** ! Tout le monde connait cette mémoire de masse, utilisées par tous comme mémoire de stockage. Non-volatile, pouvant contenir beaucoup de données, cette mémoire s'est imposée partout où l'on avait besoin d'elle. Ce soir, nous allons entrer dans l'intimité de cette star, voir ce qu'il peut y avoir dedans, et on va se rendre compte qu'un disque dur, c'est quand même franchement plus simple qu'une mémoire RAM (du moins, en apparence).

C'est fait en quoi ?

Un disque dur est tout de même quelque chose d'assez rempli, et on trouve beaucoup de composants divers et variés : des plateaux, de l'électronique de commande, des moteurs, etc.

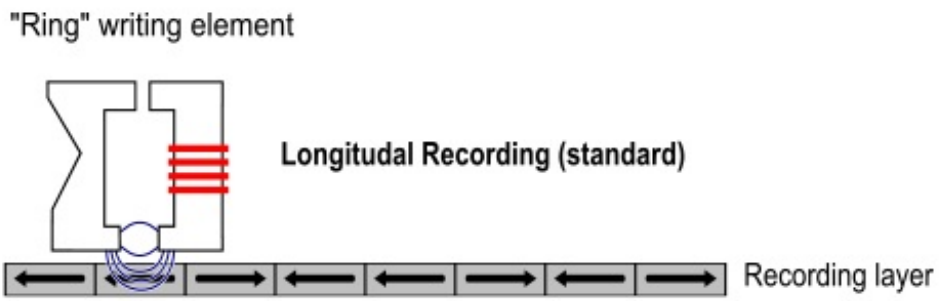
Et si on ouvrait un peu notre disque dur pour voir ce qu'il y a dedans ?

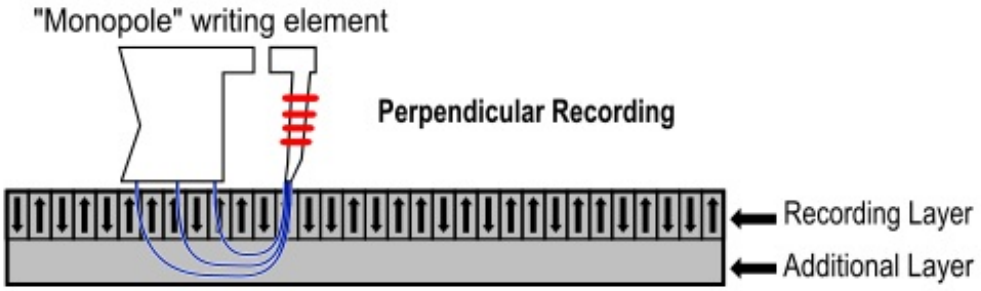
Plateaux

Ce disque dur est composé de plusieurs **plateaux**, fabriqués dans un matériau magnétique et sur lesquels on inscrit des données. Ces plateau sont composés d'une espèce de plaque, fabriquée dans un matériau peu magnétisable, recouvert de deux couches de matériau magnétique : une couche sur chaque face. Chacune de ces couches de matériau magnétique est découpée en petits blocs de données, chacun capable de contenir un bit. Sur les anciens disques durs, le stockage d'un bit dans une de ces cellules est très simple : il suffit d'aimanter la cellule dans une direction pour stocker un 1 et dans l'autre sens pour stocker un 0.

Les nouveaux disques durs fonctionnent sur un principe légèrement différent. Les disques durs récents utilisent deux cellules pour stocker un bit. Si ces deux cellules sont aimantées dans le même sens, c'est un zéro, et c'est un 1 sinon. Les disques durs basés sur ce principe permettent de stocker plus de données à surface égale. Cela vous paraitra surement bizarre, mais il faudra me croire sur parole. Expliquer pourquoi serait assez compliqué, et je ne suis pas sur que parler de Giant Magneto Resistance vous aiderais.

Quoiqu'il en soit, la façon dont on aimante ces blocs diffère suivant le disque dur. Certains aimantent ces blocs à la verticale, et d'autres à l'horizontale. Pour simplifier, on va simplement dire que pour des raisons techniques, les disques durs récents utilisent l'aimantation verticale. Cela permet de prendre moins de place pour stocker un bit, et donc d'avoir des disques durs contenant plus de données pour la même taille.





Petite remarque pour ceux qui n'auraient pas remarqués : les deux faces d'un plateau sont utilisées pour stocker des données. Ces plateaux entourent un axe central autour duquel les plateaux vont tourner. Plus ces plateaux tournent vite, plus le disque dur sera rapide.

Les disquettes fonctionnent sur un principe semblable à celui du disque dur, à une différence près : il n'y a qu'un seul plateau.

Têtes de lecture/écriture

Notre disque dur contient aussi de petits dispositifs mobiles capables de lire ou écrire une donnée sur le disque dur : les **têtes de lecture – écriture**.

Chacune de ces têtes de lecture-écriture est un dispositif assez simple. Il s'agit d'un espèce de bras mécanique dans lequel passe un fil électrique. Ce fil électrique affleure légèrement au bout de ce bras en formant une espèce d'hélice, formant un petit électroaimant qui va servir à lire ou écrire sur le plateau. Lorsque l'on veut écrire, il suffira d'envoyer un courant électrique dans le fil de notre tête de lecture : cela créera un champ magnétique autour de l'électroaimant qui est au bout du bras, ce qui permettra d'aimanter le plateau. Pour lire, il suffira d'approcher la tête de la cellule à lire : le champ magnétique de la cellule aimanté va alors créer une tension dans notre électroaimant, qui se propagera dans le fil et qu'on pourra interpréter comme un zéro (tension normale) ou un 1 (tension plus élevée que prévu).

Ces têtes de lecture se déplacent au-dessus des plateaux, et sont entrainées par un moteur capable de les faire tourner autour des plateaux : cela permet de déplacer les têtes de façon à les placer au dessus des données à lire ou écrire. A l'arrêt, les têtes de lecture sont rangées bien sagement dans un emplacement bien particulier : pas question de les arrêter sur place ! Si une tête de lecture-écriture touche la couche magnétique, alors l'endroit sur lequel la tête de lecture-écriture a atterri est définitivement endommagé.

On trouve entre une et deux têtes de lecture-écriture pour chaque plateau : généralement, on trouve une tête de lecture sur chaque face, pour pouvoir lire et écrire sur les deux faces d'un plateau. Pour s'y retrouver et choisir quelle tête de lecture-écriture utiliser, celles-ci sont numérotées par un numéro unique qui permet de les identifier.

Électronique de commande

Positionner nos têtes de lecture juste au-dessus de la cellule mémoire à lire ou écrire ne se fait pas comme par magie. Pour cela, il faut commander les moteurs qui entrainent les plateaux et les têtes de lecture de façon à ce que les têtes se positionnent correctement. Pour cela, on trouve divers circuits électroniques qui sont chargés de calculer quelles sont les tensions à envoyer aux moteurs de façon à faire accélérer ou décélérer nos têtes correctement. Notre disque dur contient aussi des circuits chargés de gérer ou de lire la tension présente dans le fil, pour effectuer des lectures ou des écritures. Et enfin, on trouve des circuits chargés de communiquer avec le bus ils reçoivent les ordres et les données en provenance du bus, et peuvent envoyer une

donnée lue depuis le disque dur sur le bus.

L'ensemble forme l'**électronique de commande** du disque dur.

Adressage d'un disque dur

Notre plateau est donc capable de stocker des données sans aucun problèmes. Reste à savoir comment sélectionner les bits et les octets inscrits sur notre plateau. Et on va voir que ce n'est pas une mince affaire : les concepteurs de disques durs ont organisé l'intérieur de notre disque dur suivant un schéma bien établi, plus ou moins nécessaire pour retrouver nos données sur celui-ci. On va donc devoir apprendre comment nos données sont organisées en pistes et secteurs sur le disque dur pour pouvoir comprendre comment sont adressées nos données.

Pistes

Ces bits sont tous regroupés sur une face de notre plateau en cercles concentriques qu'on nomme des **pistes**.

Toutes ces pistes contiennent le même nombre de bits ! Cela peut paraitre bizarre, mais il y a une explication à cela. Au bord du disque, la taille d'une piste est censée être plus grande qu'au centre : on devrait pouvoir y stocker plus de bits. Sauf que notre plateau tourne, et que la vitesse des bords est plus grande à la périphérie qu'au centre. Vu que lire ou écrire prend toujours le même temps, on est obligé de stocker nos bits sur une surface plus grande pour éviter de changer de bits en plein milieu d'une lecture/écriture parce que le plateau tourne trop vite.

Ces pistes sont toutes numérotés dans une face d'un plateau : chaque face contenant n pistes, chaque piste aura un numéro compris entre 1 et n. Mais attention : deux pistes peuvent avoir le même nombre si celles-ci sont sur des faces ou des plateaux différents. Ces pistes ayant le même numéro seront alors à la verticale les unes des autres : elles formeront ce qu'on appelle un **cylindre**.

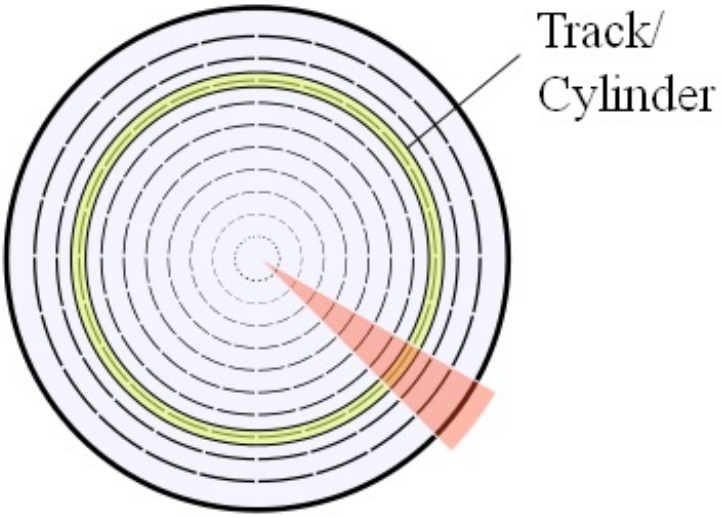
Secteurs

Ces pistes sont découpées en blocs de taille fixe qu'on appelle des **secteurs**. Quand on veut lire ou écrire sur notre disque dur, on est obligé de lire ou d'écrire l'intégralité de notre secteur. Pour simplifier le travail de l'électronique du disque dur, on préfère utiliser des paquets ayant une taille de la forme 2^n. Sur les disques durs actuels, un secteur a une taille de 512 octets, soit 4 096 bits.

Ces secteurs contiennent des données, mais pas seulement. Le début de chaque secteur est identifié par un préambule, qui permet de délimiter le secteur sur une piste. Ce préambule est suivi des données du secteur proprement dit, puis de bits de correction d'erreur, qui servent à détecter et corriger d'éventuelles corruptions de données du secteur.

Contenu d'un secteur

Toutes les pistes contiennent le même nombre de secteurs, et chaque secteur d'une piste est numéroté : cela permet de retrouver un secteur en particulier dans une piste. Ainsi, si une piste possède n secteurs, chacun de ces secteur est numéroté de 1 à n. Mais attention : deux secteurs peuvent avoir le même numéro si ceux-ci sont sur des plateaux ou sur des pistes distinctes.



Sur ce schéma, le machin rouge représente tous les secteurs ayant le même numéro sur le même plateau, et sur des pistes différentes.

Adressage CHS

Pour localiser un secteur sur un disque dur, il suffit de préciser le plateau, la face de celui-ci, le numéro de la piste (en fait, c'est le numéro du cylindre, mais passons) et le numéro du secteur. Ainsi, chaque secteur possède une adresse composée des numéros de la tête de lecture (qui sert à identifier la plateau et la bonne face), de piste et de secteur vus plus haut : c'est ce qu'on appelle l'**adresse CHS**.

Historiquement, nos ordinateurs utilisaient 10 bits pour coder le numéro de piste, 8 bits pour la tête de lecture (parfois 4) et 6 bits pour le numéro de secteur : nos adresse CHS étaient limitées à 24 bits. Cela limitait la taille maximale possible du disque dur à environ 500 mébi-octets. Pour contrer cette limite, on a inventé diverses astuces.

Une de ces astuces consiste à transformer les coordonnées CHS codées sur 24 bits (10 bits pour la piste + 8 pour le plateau + 6 pour le numéro de secteur) en coordonnées de 28 bits.

Adressage LBA

Avec la progression de la taille des disques durs, on a inventé l'**adressage LBA**. Celui-ci numérote simplement chaque secteur du disque dur par un nombre, sans se préoccuper de son numéro de tête, de cylindre ou de secteur. Il est donc identifié par un simple nombre : l'adresse LBA, qui peut être traduite en une adresse CHS codée sur suffisamment de bits pour pouvoir adresser toutes les cellules de notre disque dur.

Requêtes d'accès au disque dur

La communication avec le disque dur se fait via un bus particulier qui dépend fortement de votre ordinateur : ce peut être un bus P-ATA, S-ATA, SCSI, etc. Quoiqu'il en soit, ce que notre processeur va envoyer sur ces bus, ce ne sont rien d'autre que des ordres, des requêtes du style : "va lire à telle adresse", ou encore "va écrire à telle adresse". Ces requêtes sont envoyées au disque dur et sont gérées par l'électronique de commande du disque dur.

Sur les disques durs anciens, on devait attendre qu'une requête soit terminée avant d'en e,voyer une autre. Vu que l disque dur est assez lent, le temps entre l'envoi de deux requêtes était assez long. Pour limiter la casse, les disques durs "récents" permettent d'envoyer de nouvelles requêtes, même si le disque dur est en train d'en traiter une autre. Ces requêtes anticipées sont alors mises en attente et commenceront à être traitée quand le disque dur en aura terminé avec la requête en cours. Pour cela, nos disques durs incorporent une sorte de mémoire dans laquelle on va stocker les requêtes en attente dans l'ordre d'arrivée. Cette *Request Queue* va aussi accumuler les requêtes anticipées, qui seront traitées dans leur ordre d'arrivée. Bien sur, cette *Request Queue* a une taille limitée : si jamais elle est pleine, le disque dur enverra un signal spécial au processeur, afin de dire à celui-ci d'arrêter d'envoyer des requêtes.

Les disques durs S-ATA récents dotés de la bonne carte mère permettent de faire quelques optimisations sur le contenu de cette *Request Queue*. Ils peuvent changer l'ordre de traitement des requêtes afin de diminuer la distance que la tête de lecture doit parcourir pour trouver la donnée. L'électronique de commande peut ainsi changer l'ordre de traitement des requêtes pour accéder le plus possible à des données proches. Au lieu de faire sans cesse des aller et retours, notre disque dur peut tenter d'accéder de préférence à des données proches dans un ordre différent. Cela s'appelle du ***Native Command And Queing***.

Enfin, dernière optimisation, nos disques durs incorporent une sorte de mémoire cache qui permet de diminuer le temps d'accès à des données accédées récemment. Ainsi, toute donnée lue (ou écrite) est placée dans cette mémoire tampon. Si le disque dur veut relire cette donnée dans un avenir proche, le disque dur n'a pas à aller relire cette donnée sur le disque dur : pas besoin de déplacer des tête de lecture et d'attendre qu'elles se mettent en place, la donnée est immédiatement disponible dans cette mémoire cache ultra-rapide.

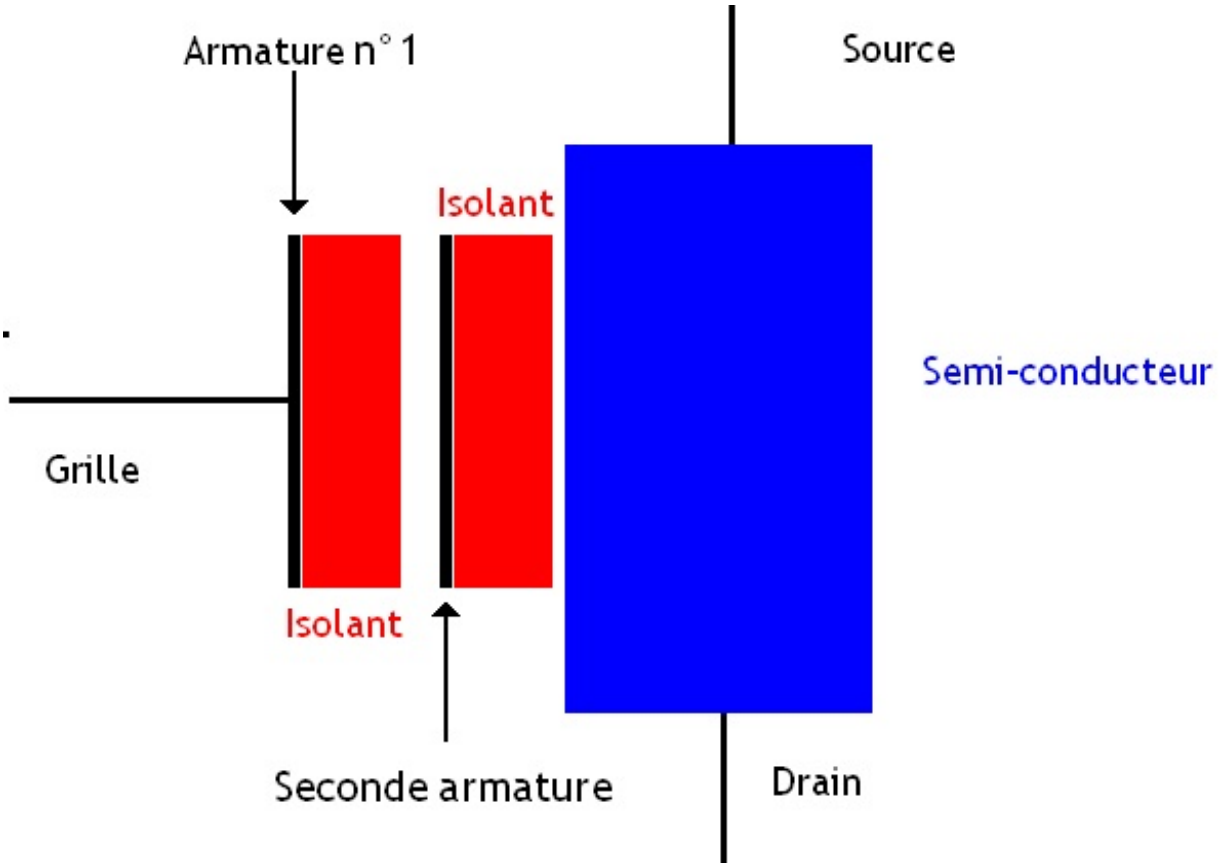
Mémoires FLASH

Les mémoires FLASH sont utilisées comme mémoires de masse un peu partout de nos jours : SSD, clés USB, BIOS, et dans des tas d'autres composants dont vous ne soupçonnez même pas l'existence. Ce sont des mémoires **EEPROM**, contrairement à ce que certains croient. En effet, pas mal de personnes croient à tort que certaines mémoires FLASH sont des mémoires RAM, ou du moins utilisent leur technologie. C'est faux, même si certaines caractéristiques de certaines de ces mémoires FLASH peuvent faire croire le contraire.

Cellule mémoire de FLASH

Mais avec quoi sont fabriquée ces mémoires FLASH ? On sait comment sont fabriquées nos bonnes vielles SRAM et DRAM, mais qu'en est-il de ces FLASH ?

Elles sont fabriquées avec des transistors. Plus précisément, on utilise un seul transistor pour fabriquer une cellule mémoire de FLASH. Mais ce transistor est un peu particulier. Il ne s'agit pas d'un bon vieux transistor MOSFET comme on en a vu au chapitre 2 : il s'agit d'un **floating gate transistor** qui possède deux armatures et deux couches d'isolant !



Comme vous le voyez sur ce schéma, on retrouve bien deux armatures en métal. C'est dans la seconde armature qu'on stockera notre bit : il suffira de la remplir d'électrons pour stocker un 1, et la vider pour stocker un 0. Ce remplissage est assez compliqué et parler d'effet tunnel ou des divers phénomènes physiques qui permettent d'écrire dans ces mémoires FLASH serait surement un peu compliqué, aussi je me permet de passer tout cela sous silence. Sachez juste que ce remplissage ou vidage se fait en faisant passer des électrons entre la grille et le drain, et en plaçant une tension sur la grille : les électrons passeront alors dans la grille en passant à travers l'isolant.

Mémoires FLASH MLC

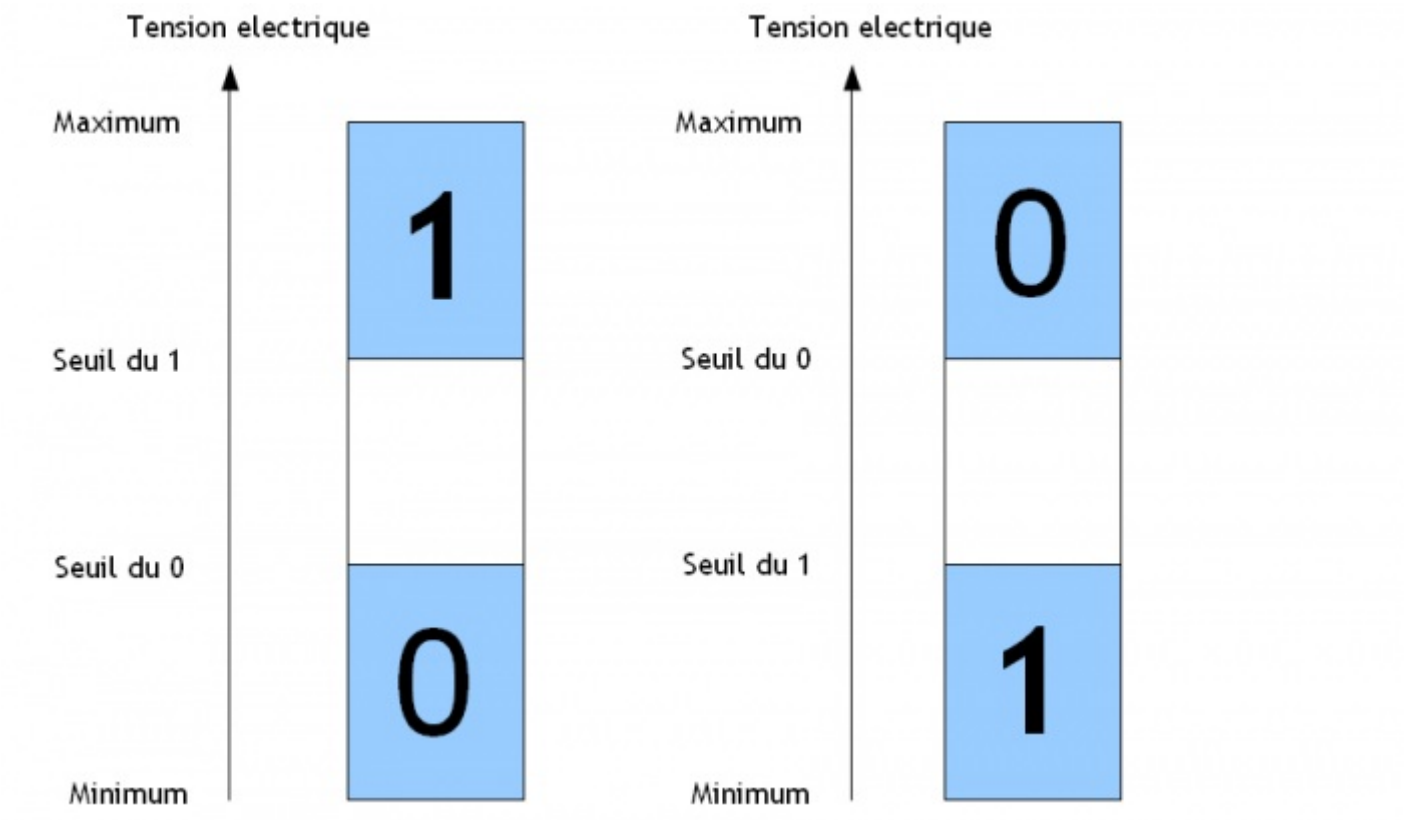
Il existe deux autres types de mémoire FLASH :

- les ***Single Level Cell***, ou SLC ;
- et les ***Multi Level Cell*** ou MLC.

S'il n'y a pas grand chose à dire sur les mémoires SLC, nos mémoire FLASH MLC sont différentes de tout ce qu'on a pu voir jusqu'à présent.

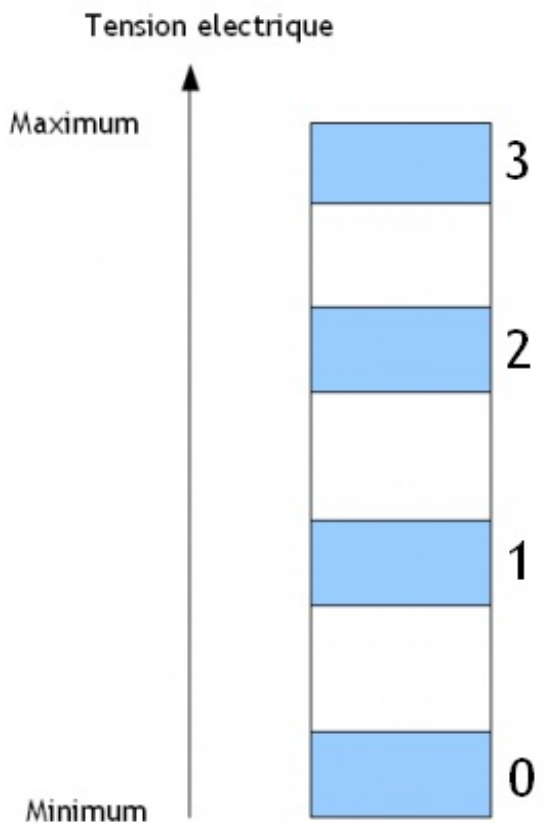
Vous vous souvenez que, dans un ordinateur, nos bits sont représentés sous la forme d'une tension avec un intervalle pour le 1, et un autre pour le zéro ?

Dans nos mémoires FLASH SLC, c'est la même chose, cette tension sera simplement celle mesurable sur notre seconde armature : elle dépendra du remplissage ou de la vacuité de l'armature.



Nos mémoires SLC fonctionnent ainsi.

Pour les mémoires FLASH MLC, c'est la même chose à un détail prêt : elles utilisent plus de deux intervalles, et peuvent ainsi stocker des informations codée en base 3 ou 4 avec une seule tension ! Une simple cellule mémoire peut ainsi stocker plusieurs bits.



Ainsi, on peut utiliser une seule cellule mémoire MLC pour stocker plusieurs bits au lieu de plusieurs cellules mémoires SLC : ça prend beaucoup moins de place !

Les mémoires FLASH ne sont pas des RAM !

Personnellement, j'ai souvent entendu dire que les mémoires FLASH étaient des mémoires RAM, ou encore qu'elles utilisaient leur technologie, et autres affirmations gratuites. Ce genre d'idées reçues est assez répandue, et quelques précisions s'imposent : non, **les mémoires FLASH n'ont rien à voir avec les mémoires RAM**, et on va expliquer pourquoi.

La différence tient dans la façon dont les mémoires FLASH vont écrire leurs données.

Programmation versus effacement

Une mémoire FLASH est divisée en blocs dont la taille varie entre 16 et 512 kibioctets. Notre mémoire FLASH est intégralement remplie de 1 par défaut. Mettre un bit à 0 est facile sur notre mémoire FLASH, et ne pose pas de problème. A coté, on trouve l'opération inverse : l'effacement qui consiste à remettre ce bit à 1. Programmer un bit individuel est parfaitement possible au niveau des circuits de notre mémoire, ce qui fait qu'on peut accéder à un octet en écriture si on met certains ou la totalité de ses bits à zéro, sans mettre aucun bit à 1. Par contre, remettre un bit individuel à 1 est impossible : on est alors obligé de réécrire tout le bloc !

Pour ce genre de raisons, on considère que les FLASH sont considérées comme des mémoires EEPROM : on ne peut pas toujours accéder à un octet en écriture sans devoir reprogrammer un gros morceau de la mémoire. Sur une mémoire RAM, on aurait pu modifier notre octet et seulement celui-ci : on est obligé de reprogrammer tout un bloc de mémoire FLASH.

Une RAM interne

Cette reprogrammation pose un gros problème : comment reprogrammer un bloc entier sans perdre son contenu ? Pour cela, le contrôleur du disque dur va utiliser une mémoire RAM interne à notre mémoire FLASH, afin d'éviter tout problème.

Voici comment se déroule la reprogrammation d'un bloc complet de mémoire FLASH :

- les circuits de la mémoire FLASH vont alors lire toute la cellule qui contient la donnée à modifier et stocker son contenu dans une petite mémoire RAM interne ;
- la donnée va être modifiée dans cette mémoire RAM interne ;
- les circuits de la FLASH vont ensuite effacer totalement la cellule de mémoire EEPROM avant de la reprogrammer avec le

contenu de la RAM interne, qui contient la donnée modifiée.

FLASH NAND et NOR

Néanmoins, ce que je viens de dire plus haut est à nuancer quelque peu. Il existe deux types de mémoires FLASH qui diffèrent par la façon dont sont reliées les cellules mémoires : les **FLASH NOR** et les **FLASH NAND** ; et ces deux types accèdent différemment à leurs données.

FLASH NOR

Ces mémoires tirent leur nom de la façon dont sont câblées leurs cellules mémoires, qui ressemble fortement au câblage d'une porte NOR constituée de transistors CMOS. Elles sont assez rapides, mais ont une mauvaise densité : leurs cellules mémoires prennent de la place et on ne peut pas en mettre beaucoup sur surface fixée.

Dans les mémoires FLASH de type NOR, chacun des octets présents dans un bloc possède une adresse. La reprogrammation ou la lecture ne posent pas vraiment de problèmes : elles peuvent se faire octets par octets. Par contre, l'effacement se fait par blocs, sans qu'on ne puisse y faire quoique ce soit.

Comme je l'ai dit, chaque bloc est adressable, ce qui fait que les FLASH NOR ressemble beaucoup aux mémoires EEPROM ou RAM courantes.

FLASH NAND

Ces mémoires tirent leur nom de la façon dont sont câblées leurs cellules mémoires, qui ressemble fortement au câblage d'une porte NAND constituée de transistors CMOS. Contrairement aux FLASH NOR, ces FLASH NAND peuvent avoir une grande capacité sans problème.

Mais par contre, l'accès est plus lent et ne se fait pas octets par octets pour le lecture ou la programmation. Pour les FLASH NAND, lecture, programmation, et effacement se font sur des morceaux de blocs ou des blocs entiers. Il faut savoir que dans ces mémoires FLASH NAND, les blocs sont eux-mêmes découpés en **pages**, d'environ 4 kibioctets, qu'on peut lire ou programmer individuellement. Mais l'effacement se fait toujours bloc par blocs.

L'accès à ces pages ou blocs n'est pas direct comme pour une FLASH NOR : tout se fait par l'intermédiaire d'une mémoire RAM interne (c'est la RAM interne vue plus haut, qui sert entre autres pour l'effacement). Toute donnée à lire ou écrire est ainsi copiée dans cette RAM avant d'être copiée sur le bus (lecture) ou dans un bloc/page (écriture).

Les SSD

Les disques durs magnétiques sont encore des composants très utilisés dans nos ordinateurs. Mais cela risque de changer à l'avenir. De nouveaux types de disques durs ont fait leur apparition il y a de cela quelques années, et ceux-ci pourraient bien remplacer nos bons vieux disques durs magnétiques. Ces SSD ne sont pas fabriqués avec des dispositifs magnétiques comme nos bons vieux disques durs, mais sont justement crées avec de la mémoire FLASH. Ce sont les ***Solid State Drive***, plus connus sous le nom de **SSD**.

Pourquoi avoir inventé ces SSD, alors que nos disques durs ne semblent pas avoir de problèmes ?

En fait, ces SSD ont plusieurs avantages qui pourraient leur permettre de prendre le pas sur leurs concurrents magnétiques.

- Ceux-ci sont des dispositifs purement électroniques : il n'y a pas de pièce mécanique en mouvement, susceptible de se casser en rendant notre disque dur inutilisable. Faites tomber votre disque dur par terre, et vous pouvez être certain que la tête de lecture-écriture sera morte. Ce qui fait que ces SSD sont plus fiables que les disques durs.
- Autre avantage : leur temps d'accès. Celui-ci est bien plus faible que le temps d'accès d'un disque dur. Avec un SSD, on n'a pas besoin de déplacer des pièces mécaniques, positionner la tête de lecture, etc : on accède à notre donnée directement, ce qui est plus rapide.
- Et enfin, dernier avantage : ils consomment beaucoup moins d'énergie.

Partie 5 : Périphériques, bus, et entrées-sorties

Un processeur et une mémoire seuls ne servent pas à grand chose. Un programme, quel qu'il soit, doit pouvoir se rendre utile, et il faut bien que celui-ci puisse finir par communiquer avec l'extérieur de l'ordinateur. Pour cela, on a inventé de nombreux périphériques, rattachés à notre ordinateur, ainsi que des bus qui vont permettre à notre processeur et notre mémoire d'échanger avec eux. Dans ce chapitre, nous allons voir un peu comment nos périphériques vont communiquer avec notre processeur.

Bus, cartes mères, chipsets et Front Side Bus

Dans notre ordinateur, tous les composants (mémoire, processeurs...), sont fabriqués séparément. Pour relier tout ces composants ensemble, on place ces composants sur un gros circuit imprimé nommé la **carte mère** sur laquelle on trouve un ou plusieurs bus pour relier le tout. En somme, une carte mère n'est donc rien qu'un gros tas de fils reliés à des connecteurs sur lesquels on va brancher des composants. Enfin presque, il y a des trucs en plus dessus : le **BIOS**, de quoi créer les signaux d'horloge servant à cadencer les périphériques, etc.

Mais ces composants ne communiquent pas que par un seul bus. Il existe un bus pour communiquer avec le disque dur, un bus pour la carte graphique, un pour le processeur, un pour la mémoire, etc. De ce fait, de nombreux bus ont étés inventés et un ordinateur "lambda", avec sa souris, son écran et son unité centrale contient un nombre impressionnant de bus.
Jugez plutôt :

- le **SMBUS**, un bus inventé par Intel en 1985 qui est utilisé pour communiquer avec les ventilateurs, les sondes de températures et les sondes de tension présentes un peu partout dans notre ordinateur : la vitesse des ventilateurs ne se règle pas toute seule comme par magie ;
- les bus **USB**, que vous connaissez tous et pour lequel je ne dirais rien sinon qu'il existe un tutoriel sur ce bus sur le Siteduzéro : **Comprendre l'USB et bricoler un périphérique**;
- le bus **PCI**, utilisé pour le cartes sons et qui servait autrefois à communiquer avec les cartes graphiques ;
- le bus **AGP**, autrefois utilisé pour les cartes graphiques ;
- le bus **PCI-Express**, utilisé pour communiquer avec des cartes graphiques ou des cartes sons ;
- le bus **P-ATA**, relié au disque dur ;
- le bus **S-ATA** et ses variantes : eSATA, eSATAp, ATAoE, utilisé pour communiquer avec le disque dur ;
- le bus **Low Pin Count**, qui permet d'accéder au clavier, aux souris, au lecteur de disquette, et aux ports parallèles et séries ;
- le bus **ISA** et son cousin le bus **EISA**, autrefois utilisé pour des cartes d'extension ;
- l'**Intel QuickPath Interconnect** et l'**HyperTransport**, qui relient les processeurs récents au reste de l'ordinateur ;
- le **FireWire** (1394) ;
- le bus **SCSI** et ses variantes (SCSI Parallel, Serial Attached SCSI, iSCSI) qui permettent de communiquer avec des disque durs ;
- le bus **MIDI**, une véritable antiquité oubliée de tous qui servaient pour les cartes sons ;
- notre fameux **RS-232** utilisé dans nos ports série ;
- et enfin le bus **IEEE-1284** utilisé pour le port parallèle.

Et encore, je crois que j'en ai oublié un ou deux !

De plus, chacun de ces bus est souvent mis à jour, et de nouvelles versions apparaissent, qui sont plus rapides, moins énergivores, etc. Par exemple, le bus PCI est décliné en 7 versions, la dernière en date datant de 2002. Autre exemple : l'USB, décliné en 4 versions (1.0, 1.1, 2.0, 3.0).

Tous ces bus sont très différents les uns des autres, et ont des caractéristiques très différentes. Pourtant, à première vue, rien de plus simple qu'un bus : c'est juste un tas de fils.

Mais alors, qu'est-ce qui peut bien les différencier ?

Un bus, c'est rien qu'un tas de fils...

Il suffit de demander :

Caractéristique	Définition
Sa largeur	C'est le nombre de bits qui peuvent être transmis simultanément sur notre bus.
Son débit	C'est le nombre de bits que notre bus peut transmettre par seconde. Plus celui-ci est élevé, plus le bus est

binaire	rapide.
Sa latence	C'est le temps d'attente que met une donnée à être transférée sur le bus. Ce temps de latence dépend de la fréquence du bus et d'autres paramétrés. Plus il est bas, plus le bus est rapide.
Son caractère *Half Duplex*, *Full duplex* ou *Simplex*	Vous verrez ce que ça signifie dans la suite du chapitre.
Son caractère synchrone ou asynchrone	Certains bus possèdent un fil sur lequel circule un signal d'horloge permettant de synchroniser les différents composants : ce sont les bus synchrones. D'autres se passent de signal d'horloge, et synchronisent leurs composant par d'autres méthodes.
Son protocole	Le protocole d'un bus définit comment celui est géré. Ce protocole définit quand et comment les données doivent être envoyées sur le bus. Mais ça ne se limite pas à ça : en effet, certains bus ont besoin de mécanismes assez sophistiqués pour fonctionner correctement. Pour donner un exemple, on peut citer le cas dans lequel plusieurs composants électroniques doivent transmettre leurs données sur un seul et unique bus. Le protocole doit alors intégrer des mécanismes permettant aux composants de ne pas se marcher sur les pieds en monopolisant le bus ou en écrivant des données en même temps.
Son bus de commande	Ceux-ci varient énormément suivant le bus : • certains se contentant d'un seul bit ; • d'autres ont besoin de beaucoup de bits pour gérer pleins de paramètres différents ; • et d'autres s'en passent. Généralement, la complexité du bus de commande est fortement influencé par le protocole utilisé pour le bus.

Et encore, on vous a épargné avec cette liste assez courte ! Suivant l'utilisation d'un bus, on choisit chacune de ces caractéristiques en fonction des besoins. Par exemple, pour relier un clavier à notre ordinateur, on l'utilisera pas le même bus que pour relier une carte graphique sur la carte mère. La carte graphique aura besoin de transférer beaucoup de données par seconde et aura besoin d'un bus rapide, contrairement au clavier qui aura d'autres besoins.

Il y a peu à dire sur la fréquence, le débit binaire et la latence d'un bus, sinon que la latence doit être la plus faible possible et la fréquence et le débit binaire élevés si on veut gagner en rapidité. Par contre, il faut préciser certaines choses importantes sur la largeur du bus.

Bus série et parallèles

Il existe des bus qui ne peuvent échanger qu'un seul bit à la fois. On appelle ce genre de bus un **bus série**.
D'autres bus peuvent échanger plusieurs bits en même temps et sont appelés **bus parallèles**.

Dans nos ordinateurs, les bus qui relient le processeur à la mémoire (ou la mémoire aux périphériques) sont des bus parallèles. Ils sont généralement divisés en trois sous-bus :

- le **bus d'adresse** par lequel les adresses transitent
- le **bus de donnée** par lequel les données s'échangent entre composants.
- le **bus de commande**, qui contient au moins le bit R/W et qui gère les échanges d'informations sur le bus.

Différence de vitesse entre bus série et parallèle

Vu qu'ils peuvent transférer plusieurs bits en une fois, les bus parallèles sont bien plus rapides que les bus séries, non ?

Ça dépend de la durée durant laquelle un bus ne peut pas changer d'état. En effet, un bus doit toujours attendre un certain moment avant d'envoyer la donnée suivante. La donnée présente sur le bus va y rester durant un moment, et sera mise à jour après un certain temps.

Pour un bus série, il s'agit du temps d'envoi d'un bit, par exemple.
Pour les bus parallèle, c'est le temps que la donnée envoyée restera sur ses fils avant d'être mise à jour par une nouvelle donnée.
Pour information, si le contenu d'un bus parallèle d'une largeur de n bits est mis à jour m fois par secondes, alors son débit binaire (le nombre de bits transmis par seconde) est de $n \times m$.

On pourrait alors croire que les bus parallèles sont plus rapides, mais ce n'est pas forcément vrai. En effet, il est difficile pour un bus parallèle de mettre à jour son contenu et d'envoyer la donnée suivante rapidement. Un bus série n'a pas ce mal : la durée de transmission d'un bit est très faible.

Pourquoi ?

Voici l'explication : Lorsque la tension à l'intérieur du fil varie (quand le fil passe de 0 à 1 ou inversement), le fil va émettre des ondes électromagnétiques qui vont aller perturber la tension dans les fils d'à coté. Il faut donc attendre que la perturbation électromagnétique se soit atténuée pour pouvoir lire le bit sans se tromper. Et ces temps d'attente limitent le nombre de changement d'état du bus effectués par seconde ! D'où un nombre de données envoyé par seconde plus faible.

Autre problème : les fils d'un bus parallèle ne sont pas tous identiques électriquement : il n'est pas rare que la résistance des fils ou d'autres propriétés électriques changent très légèrement d'un fil à l'autre. Conséquence : un bit va se propager d'un bout à l'autre d'un fil à des vitesse qui varient suivant le fil. Et on est obligé de se caler sur la vitesse la plus basse pour éviter des problèmes techniques à la réception.

Un bus série n'a pas ce genre de problèmes et peut donc envoyer un grand nombre de bits très rapidement. Cela peut compenser le fait qu'un bus série ne peut envoyer qu'un bit à la fois assez facilement.

Dans nos PC

Je suis certains que vous connaissez l'USB, au moins de nom. Mais savez-vous ce que c'est ? Il s'agit d'une norme, qui définit un bus, le bus USB. Ce bus est un bus série, un peu particulier.

Notre bus USB définit 4 fils, notés 1, 2, 3 et 4 dans le schéma suivant.

Numéro du fil	Fonction	Contenu	Couleur
1	Tension d'alimentation	Tension égale à +5 volts	Rouge
2	D+	Donnée	Vert
3	D-	Donnée	Blanc
4	Masse	0 volts	Noir

De ce qu'on voit de ce bus, c'est qu'il possède une masse et un fil d'alimentation, ce qui permet d'alimenter le périphérique qui est connecté sur le port USB. Ben oui, votre souris USB, elle n'est pas reliée au 220 volts, et doit bien être alimentée en électricité quelque part !

On pourrait croire qu'un bus ne pouvant transmettre qu'un seul bit à la fois ne contient qu'un seul fil pour transmettre les donnée, mais ce bus en est un parfait contre-exemple : celui-ci possède 2 fils pour transmettre le bit en question.

Le truc, c'est que notre donnée n'est pas codée en utilisant un codage NRZ. On ne va pas rentrer dans les raisons qui ont poussées les créateurs de l'USB a faire ce choix. Tout ce qu'il faut savoir, c'est que notre bus USB transmet un 1 en mettant une tension de -5 volts sur la broche D- (la broche D+ contient alors un joli petit zéro volt), et transmet un 0 en mettant une tension de +5 volts sur la broche D- (la broche D- contient alors un joli petit zéro volt).

Autre exemple de bus série que vous connaissez surement : le bus S-ATA. Celui-ci sert à communiquer avec nos disques durs. Il a été inventé pour remplacer le bus P-ATA, un bus plus ancien, servant lui aussi à communiquer avec nos disques durs, mais qui avait un défaut : c'était un bus parallèle.

Simplex, Half duplex ou Full duplex

Un autre paramètre important concernant nos bus est le sens des transferts de donnée. Pour expliquer cela, on va devoir clarifier quelques notions assez simples. Vous étés prêt ? On commence !

Un composant qui envoie une donnée sur le bus est appelé un **émetteur**
Ceux qui se contentent de recevoir une donnée sur le bus sont appelés **récepteur**.

Simplex

Sur un bus simplex, les informations ne vont que dans un sens. On a donc deux cas :

- Soit les informations sont transmises d'un périphérique vers un autre composant (par exemple le processeur). L'autre sens est interdit !
- Soit les informations sont transmises d'un composant vers le périphérique. L'autre sens est interdit !

Il n'y a qu'un seul des composants qui puisse être émetteur et l'autre reste à tout jamais récepteur.

Half-duplex

Sur un bus half-duplex, Il est possible d'être émetteur ou récepteur, suivant la situation. Par contre, impossible d'être à la fois émetteur et récepteur.

Full-duplex

Un bus full duplex permet d'être à la fois récepteur et émetteur. L'astuce des bus full duplex est simple : il suffit de regrouper deux bus simplex ensemble ! Il y a un bus pour l'émission et un pour la réception.
Ces bus sont donc plus rapides, vu qu'on peut émettre et recevoir des données en même temps, mais nécessitent plus de fils à câbler, ce qui peut être un désavantage.

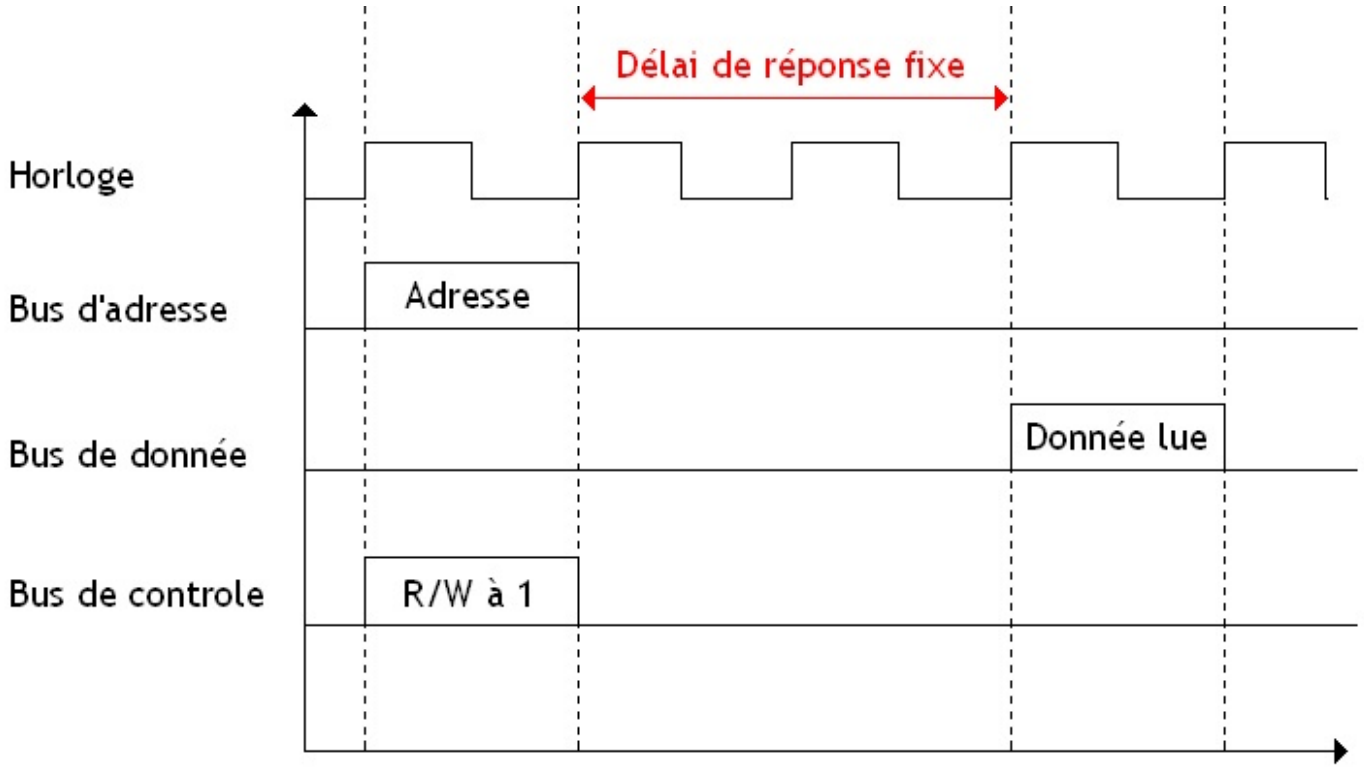
Bus synchrones et asynchrones

Bus synchrones

Certains bus sont synchronisés sur un signal d'horloge : ce sont les **bus synchrones**. Sur de tels bus, un fil est spécialement dédié à l'horloge, cette fameuse tension périodique vue il y a quelques chapitres. Quand à nos composants, ils sont reliés au bus via des bascules, afin de synchroniser les lectures/écritures des composants sur le bus.

Avec ce genre de bus, le temps de transmission d'une donnée sur le bu est fixé une fois pour toute. Ainsi, le composant qui cherche à effectuer une lecture ou un écriture sait combien de cycles d'horloge sa demande va prendre.

Exemple avec une lecture



L'avantage de ces bus est que le temps que va mettre un récepteur ou un émetteur pour faire ce qu'on lui demande est fixé une bonne fois pour toute. Par contre, ces bus posent des problèmes quand on augmente la fréquence ou la longueur des fils du bus : notre signal d'horloge va mettre un certain temps pour se propager à travers son fil, ce qui induit un léger décalage entre les composants. Si ce décalage devient trop grand, nos composants vont rapidement se désynchroniser : il faut que ce décalage soit très petit comparé à la période de l'horloge.

Plus on augmente la longueur des fils, plus l'horloge mettra de temps à se propager d'un bout à l'autre du fil qui lui est dédié, et plus ces décalages deviendront ennuyeux. Et la fréquence pose un problème similaire : plus on augmente la fréquence, plus cette période diminue comparé au temps de propagation de l'horloge dans le fil, et plus ces décalages risquent de poser problème. Mine de rien, ce genre de phénomènes fait qu'il est très difficile d'atteindre des fréquences de plusieurs centaines gigahertz sur les processeurs actuels : le fil d'horloge est trop long pour que ces décalages soient négligeables.

Double Data Rate

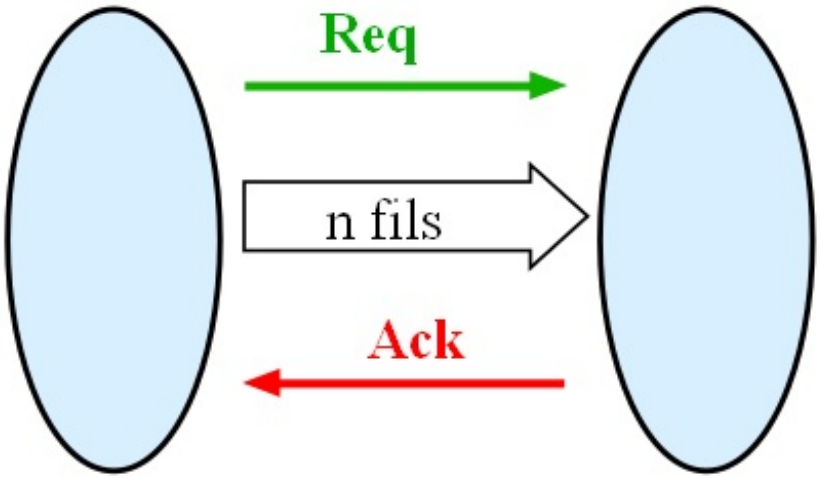
Sur certains bus, le contenu du bus n'est pas mit à jour à chaque front montant, ou à chaque front descendant, mais au deux : fronts montants et descendants. De tels bus sont appelés des bus **double data rate**.

Le but ? Cela permet de transférer deux données sur le bus (une à chaque front) en un seul cycle d'horloge : le débit binaire est doublé sans toucher à la fréquence du bus. Pour information, les bus mémoires qui relient des mémoires DDR1 au processeur sont de ce type : ils peuvent réellement transférer deux données par cycle d'horloge.

Bus asynchrones

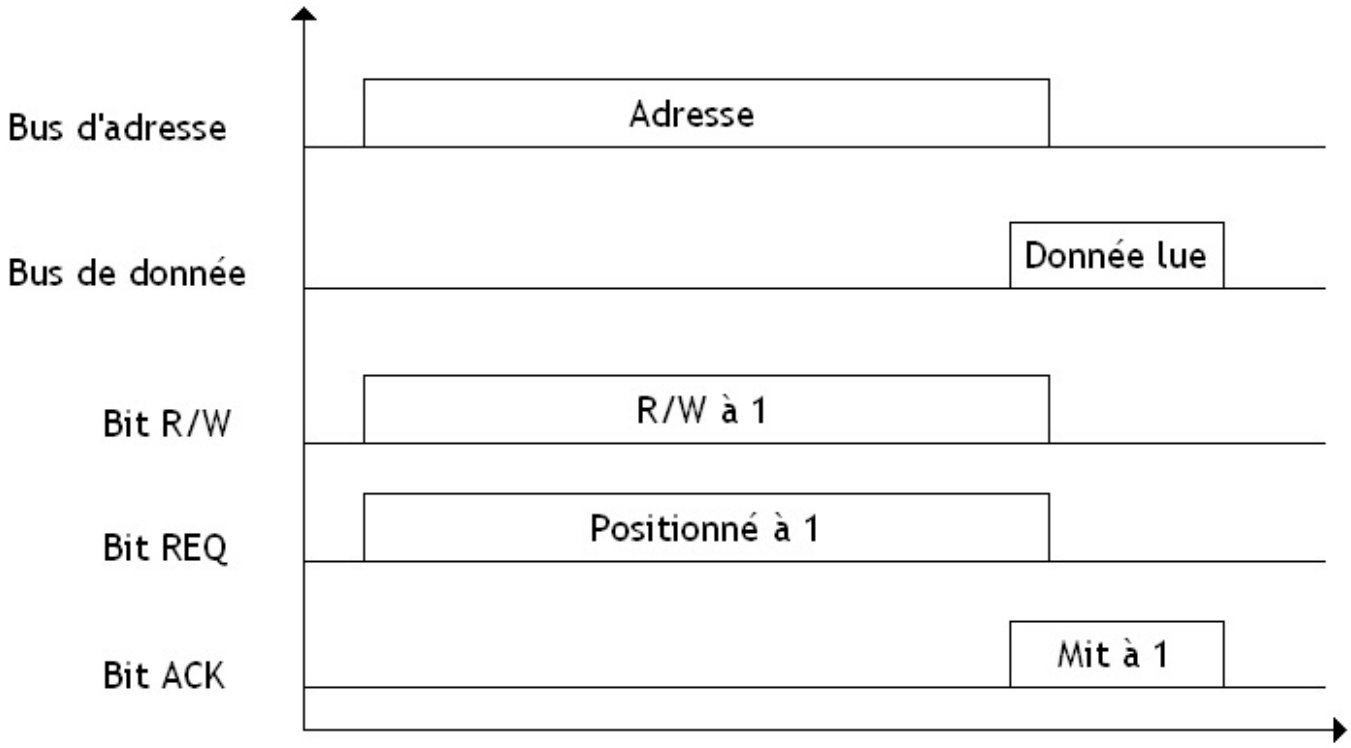
Et maintenant, j'ai une révélation à vous faire : certains bus se passent complètement de signal d'horloge.

Ces bus ont un protocole conçu spécialement pour faire communiquer deux périphériques/composants sans les synchroniser sur une horloge. Pour cela, ces bus permettent à deux composants de se synchroniser grâce à des fils spécialisés du bus de commande, qui transmettent des bits particuliers. Généralement, ce protocole utilise deux fils supplémentaires : REQ et ACK.



Lorsqu'un composant veut envoyer une information sur le bus à un autre composant, celui-ci place le fil REQ à 1, afin de dire au récepteur : "attention, j'ai besoin que tu me fasse quelque chose". Les autres composants vont alors réagir et lire le contenu du bus. Le composant à qui la donnée ou l'ordre est destiné va alors réagir et va faire ce qu'on lui a demandé (les autres composants se rendorment et se déconnectent du bus). Une fois qu'il a terminé, celui-ci va alors positionner le fil ACK à 1 histoire de dire : j'ai terminé, je libère le bus !

Exemple avec une lecture



Ces bus sont très adaptés pour transmettre des informations sur de longues distances (plusieurs centimètres ou plus), ou pour communiquer simplement avec des composants au besoin, sans avoir à les synchroniser en permanence. Sans compter qu'ils sont parfois plus rapides : on n'a pas attendre un délai fixé avant de recevoir le résultat d'une lecture/écriture. Dès que la demande est effectuée, on libère le bus. Sur un bus synchrone, on aurait eu besoin d'attendre la fin du délai fixé par le protocole du bus, même si le récepteur/émetteur a fait ce qu'il lui était demandé.

Va falloir partager !

Comme je l'ai dit plus haut, un bus est un ensemble de fils qui relie plusieurs composants. Mais le nombre de composants connectés au bus est variable suivant le bus.

Certains bus se contentent de connecter deux composants entre eux, pour leurs permettre de communiquer. Le bus n'a pas besoin d'être partagé entre plusieurs composants, et est réservé à ces deux composants. Ce sont les **bus dédiés**.

Sur d'autres bus, on peut connecter un nombre plus important de composants, qui peut être assez élevé dans certaines situations : le bus doit donc être partagé entre ces composants. Ce sont les **bus multiplexés**.

Conflit d'accès

Si on câble plusieurs composants sur le même bus, rien n'empêche ces deux composants de vouloir envoyer ou recevoir une donnée sur le bus en même temps. C'est ce qu'on appelle un **conflit d'accès au bus**. Cela pose problème si un composant cherche à envoyer un 1 et l'autre un 0 : quand on envoie plusieurs bits en même temps, tout ce que l'on reçoit à l'autre bout du fil est un espèce de mélange incohérent des deux données envoyées sur le bus par les deux composants. En clair, ça ne ressemble plus à rien à l'autre bout du fil !

Pour résoudre ce petit problème, il faut obligatoirement répartir l'accès au bus de façon à ce qu'un seul composant utilise le bus à la fois. Ainsi, chaque composant va pouvoir envoyer des données sur le bus et démarrer une communication à tour de rôle. Les composants ne pourront donc pas toujours émettre quand ils le souhaitent : si un composant est déjà en train d'écrire sur le bus, et qu'un autre veut l'utiliser, on devra choisir lequel des deux pourra émettre ses données sur le bus. Ce choix sera effectué différemment suivant le protocole du bus et son organisation, mais ce choix n'est pas gratuit : certains composants devront attendre leur tour pour avoir accès au bus. Ces temps d'attente ralentissent quelque peu les transferts. Un bus dédié n'a pas ce problème, et est donc plus rapide, plus simple.

Pour limiter les problèmes dus à ces temps d'attente, les concepteurs de bus ont inventés diverses méthodes pour gérer ces conflits d'accès, et choisir le plus équitablement et le plus efficacement possible quel composant peut envoyer ses données sur le bus. C'est ce qu'on appelle l'**arbitrage du bus**. Divers types d'arbitrage existent.

Dans l'arbitrage centralisé, un circuit spécialisé s'occupe de l'arbitrage du bus, et aucun composant n'a quoique ce soit à dire : ils doivent se contenter d'obéir aux ordres du circuit chargé de gérer l'arbitrage. Dans l'arbitrage distribué, chaque composant se débrouille de concert avec tous les autres pour éviter les conflits d'accès au bus : les composants sont reliés par des fils entre eux, et chacun décidé d'émettre ou pas suivant l'état du bus. Dans ce qui va suivre, on va vous montrer deux exemples de méthodes d'arbitrages du bus particulièrement simples.

Arbitrage par multiplexage temporel

Cet arbitrage peut se résumer en une phrase : **chacun son tour** ! Avec cet arbitrage, chaque composant a accès au bus à tour de rôle durant un temps fixe. Cet arbitrage est très simple : on laisse le bus durant quelques millisecondes à un composant avant de passer au suivant et ainsi de suite avant de revenir au point de départ.

Cet arbitrage est néanmoins peu adapté aux cas pour lesquels certains composants effectuent beaucoup de transactions sur le bus et d'autres très peu : la répartition de l'accès au bus est fixe et ne tient pas compte du fait que certains composants utilisent peu le bus et d'autres beaucoup : tous ont la même part.

Arbitrage par requête

Mais rassurez-vous, il y a moyen de faire nettement mieux et plus simple comme méthode d'arbitrage du bus : **premier arrivé, premier servi** ! L'idée est que tout composant peut accéder au bus si celui-ci est libre et se le réserver. Par contre, si jamais le bus n'est pas libre, le composant qui souhaite accéder au bus doit attendre que celui qui utilise le bus le libère.

Certains protocoles d'arbitrage ont amélioré le principe de base. Avec ces protocoles, il est possible de libérer le bus de force, et interrompre brutalement une transmission pour laisser la place à un autre composant. Sur certains bus, certains composants sont prioritaires, et les circuits chargés de la gestion du bus (peut importe leur localisation), vont pouvoir libérer de force le bus si jamais un composant un peu plus prioritaire veut utiliser le bus. D'autres font en sorte de préempter le bus, c'est à dire qu'ils donnent l'accès au bus à un composant durant un certain temps fixe. Si le composant dépasse ce temps fixe, la transmission est interrompue pour laisser la place à un autre composant. Par contre, un composant qui n'utilise pas totalement le temps qui lui est attribué peut libérer le bus prématurément pour laisser la place à un de ses camarades.

Implémentation

C'est bien beau d'avoir crée un protocole d'arbitrage du bus un peu mieux, mais encore faut-il que nos composants puissent savoir que le bus est occupé pour que ce protocole puisse fonctionner. Pas de panique : le bus de commande est là pour çà ! Il suffit de lui rajouter un fil qui sert à indiquer que le bus est occupé et qu'un composant l'utilise : le fil **Busy**.

Cet arbitrage peut être implanté aussi bien en une version centralisée qu'en une version distribuée.

Chipset, back-plane bus, et autres

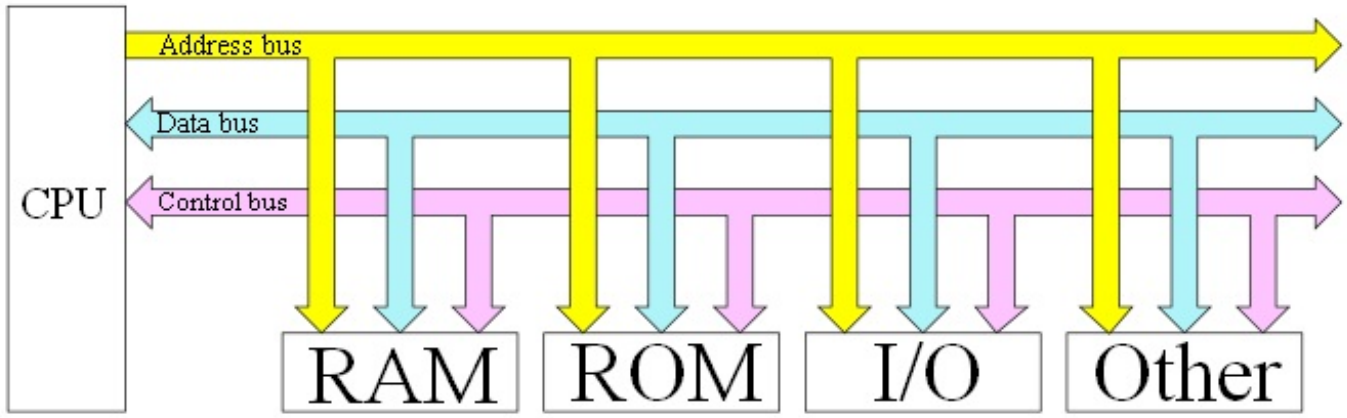
L'organisation des bus de nos ordinateurs a évoluée au cours du temps pendant que de nombreux bus apparaissaient.

On considère qu'il existe deux générations de bus bien distinctes :

- une première génération avec un bus unique, la plus ancienne ;
- une seconde génération avec des bus segmentés .

Première génération

Pour les bus de première génération, un seul et unique bus reliait tous les composants de notre ordinateur. Ce bus s'appelait le **bus système** ou *backplane bus*.



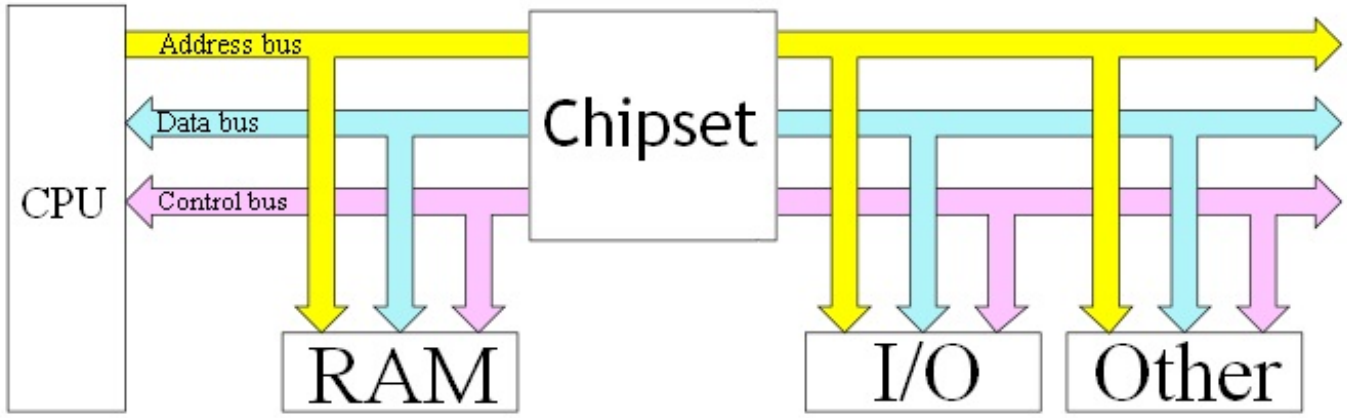
Ce bus était partagé entre tous les composants : chacun d'entre eux monopolisait le bus durant un moment, et le libérait quand il avait fini de transmettre des données dessus.

Ces bus de première génération avaient le fâcheux désavantage de relier des composants allant à des vitesses très différentes : il arrivait fréquemment qu'un composant rapide doive attendre qu'un composant lent libère le bus. Le processeur était le composant le plus touché par ces temps d'attente.

Seconde génération

Pour régler ce genre de problèmes, on a décidé de diviser le bus système en deux bus bien séparés : un bus pour les périphériques lents, et un autre pour les périphériques rapides. Deux composants lents peuvent ainsi communiquer entre eux sans avoir à utiliser le bus reliant les périphériques rapides (et vice-versa), qui est alors utilisable à volonté par les périphériques rapides. Les composants rapides et lents communiquent chacun de leur coté sans se marcher dessus.

Ces deux bus étaient reliés par un composant nommé le ***chipset***, chargé de faire la liaison et de transmettre les données d'un bus à l'autre.

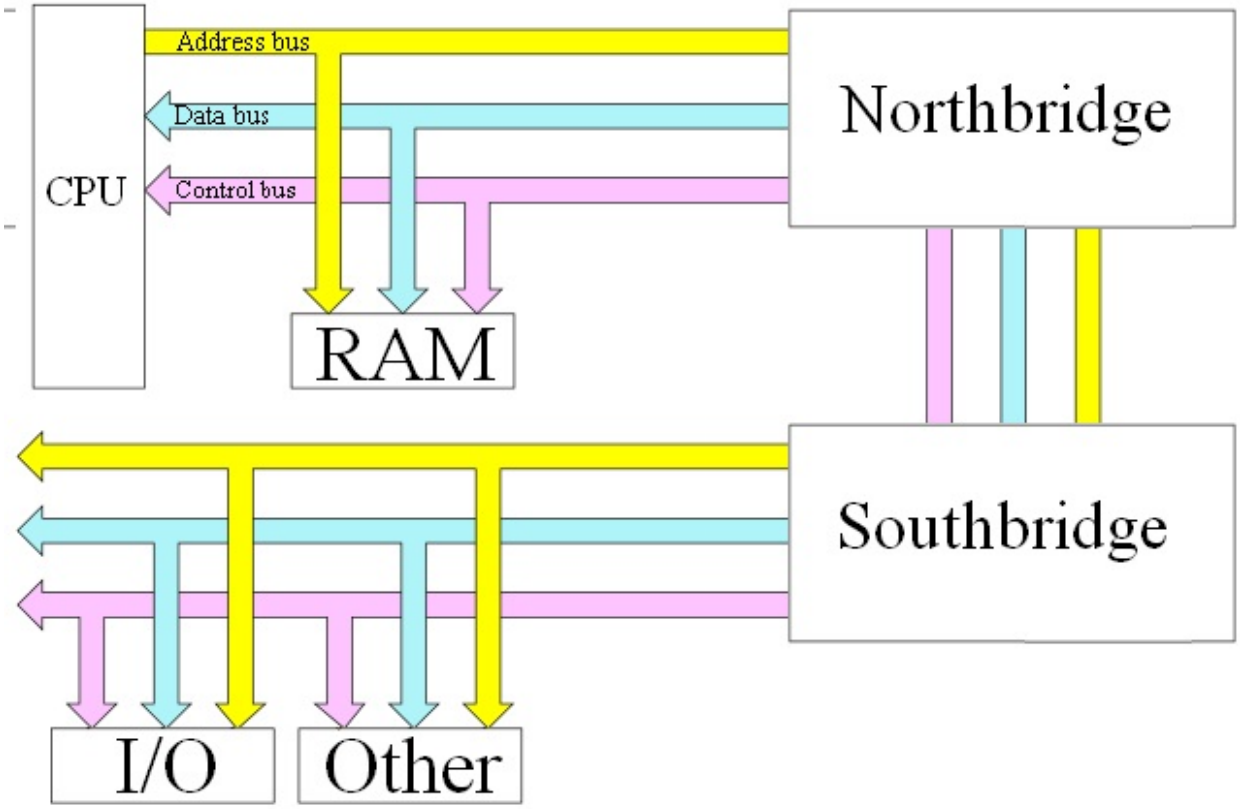


Comme vous le voyez sur cette image, les composants considérés comme rapides sont le processeur et la mémoire. Ceux-ci sont souvent associés à la carte graphique. Le reste des composants est considéré comme lent.

Northbridge et southbridge

Sur certains ordinateurs, le *chipset* est divisé en 2 :

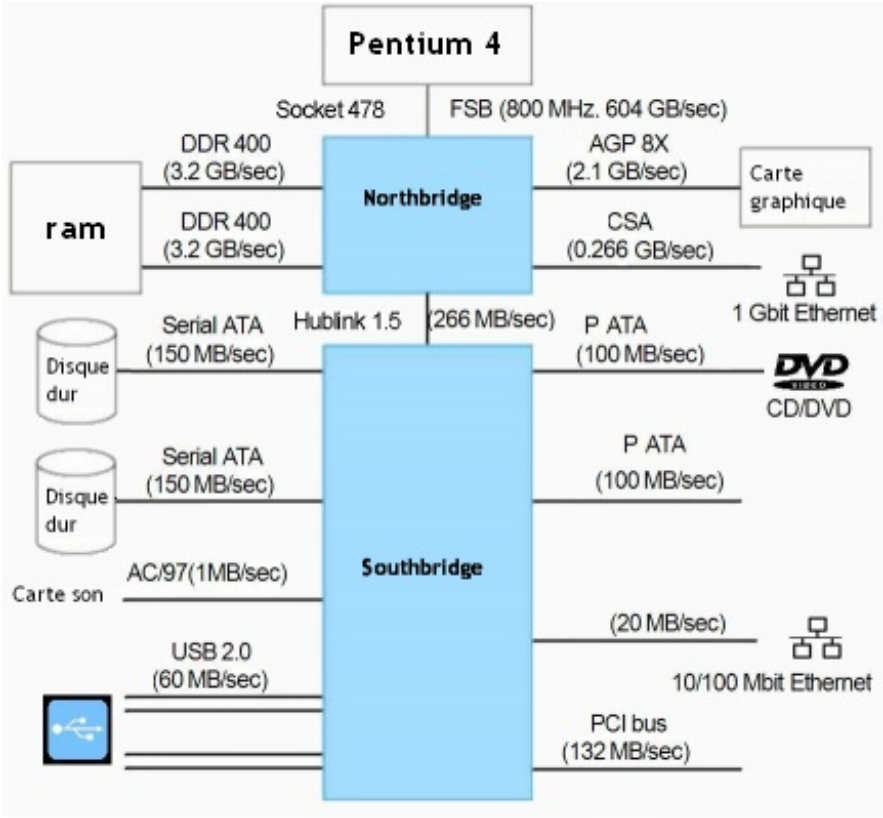
- Le ***northbridge*** :
  une partie qui s'occupe de tous les bus reliés aux composants rapides : processeurs, RAM, carte graphique (et oui !) ;
- Le ***southbridge*** :
  et une partie qui s'occupe de gérer les bus des périphériques lents, comme les disque durs, les ports USB, le clavier, etc.



De nos jours

De nos jours, nos ordinateurs contiennent bien plus que deux bus, et presque chaque composant ou contrôleur de périphérique est connecté sur le *chipset* par un bus.

Le preuve par l'exemple :



Sur les ordinateurs ayant des processeurs récents (2009-2010), le *northbridge* a disparu : celui-ci est intégré directement dans le processeur.

Front side bus

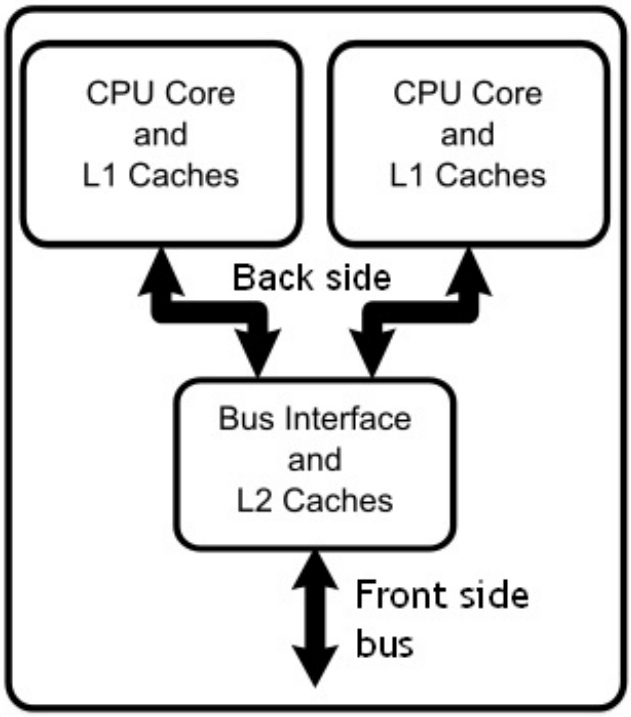
Parmi tout ces bus, un bus fait l'objet d'une attention particulière : le ***Front side bus***. C'est le bus qui relie le processeur au *chipset*. Plus celui-ci est rapide, moins le processeur a de risque d'être ralenti par la mémoire et les opérations de communications avec le *chipset*.

Ce *Front side bus*, ou FSB, est cadencé à une fréquence qui est considéré comme l'horloge de base, de laquelle toutes les autres fréquences découlent. Ainsi, la fréquence du processeur est un multiple de la fréquence du FSB : on obtient la fréquence du processeur en multipliant la fréquence du FSB par un coefficient multiplicateur. Mais le processeur n'est pas le seul à avoir ce luxe : toutes les autres fréquences de notre ordinateur sont déduite de la fréquence du FSB par d'autres coefficients multiplicateurs.

Sur certaines cartes mères, il est possible de modifier la fréquence du FSB et/ou les coefficients multiplicateurs : on peut ainsi faire varier la fréquence de notre processeur ou de notre mémoire. Certains préfèrent l'augmenter pour avoir un processeur ou une mémoire plus rapide, et font ce qu'on appelle de l'*overcloking* (surfréquenceage en français). D'autre pratiquent l'*undercloking*, et diminuent la fréquence de leur processeur pour que celui-ci chauffe moins et pour gagner quelques euros en faisant baisser leur facture EDF. Mais il s'agit là de pratiques très dangereuses qui peuvent faire rendre l'âme prématurément à votre processeur : il s'agit de manipulations qui doivent être faites par des professionnels qualifiés et des personnes qui savent ce qu'elles font.

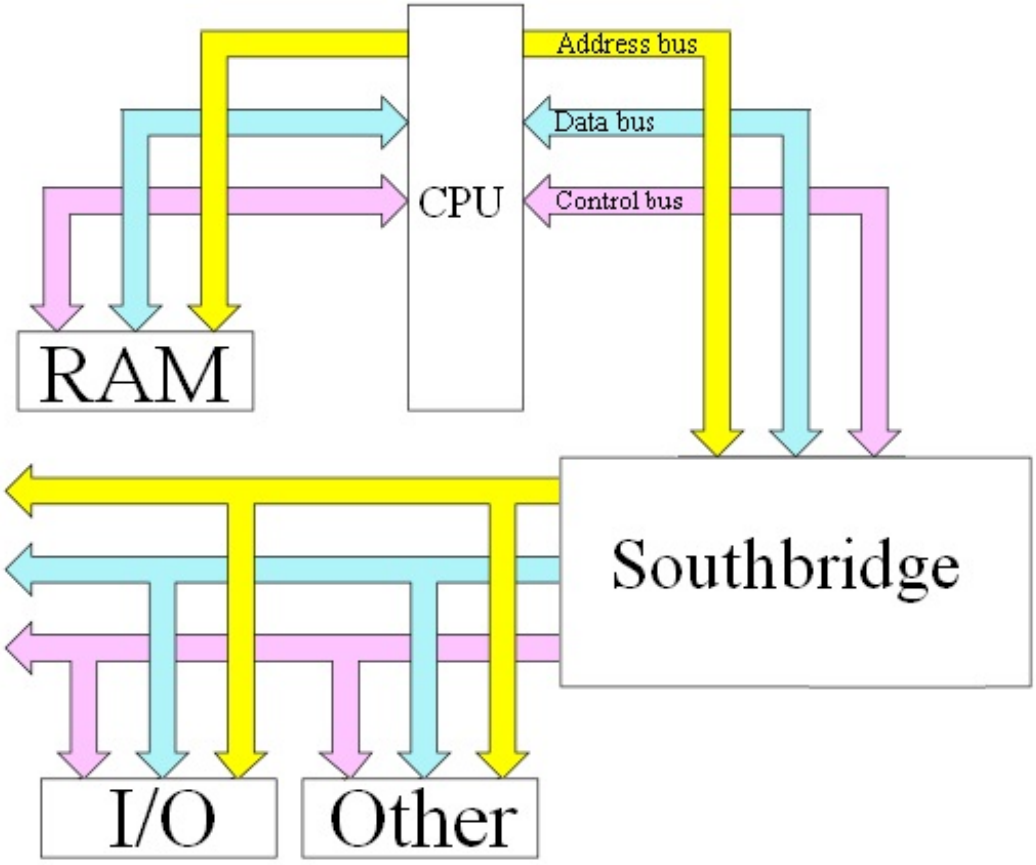
Back side bus

Moins connu que le le FSB, il existe un autre bus qui avait autrefois son importance sur certaines carte méres : le ***Back Side Bus***. Autrefois, la mémoire cache était (totalement ou en partie) séparée du processeur : le *Back Side Bus* était chargé de relier le processeur à la mémoire cache.



Architectures sans *Front side bus*

Récemment, le FSB a subit quelques évolutions. Sur les cartes mères récentes, le *northbridge* est en effet intégré au processeur, ce qui a nécessité quelques modifications. L'organisation des bus sur nos ordinateurs réels ressemble plus au schéma qui suit.



En clair, on a un bus qui relie le processeur à la mémoire, et un autre qui relie le processeur au *southbridge*, avec parfois un autre bus pour la carte graphique, mais passons. Ces nouveaux bus reliant le processeur au *southbridge* vous sont peut-être connus si vous allez souvent sur des sites parlant de l'actualité du *hardware* : il ne s'agit ni plus ni moins que des bus Intel QuickPath

Interconnect de chez Intel, et l'HyperTransport d'AMD.

Communication avec les Entrées-Sorties

Dans ce chapitre, on va voir comment nos périphériques vont faire pour communiquer efficacement avec notre processeur ou notre mémoire. Et on va voir que la situation est plus compliquée qu'il n'y parait. On a beaucoup parlé de mémoire et de processeur ces derniers chapitres, ce qui me pousse à commencer ce chapitre par quelques rappels.

On sait déjà que nos entrées-sorties (et donc nos périphériques) sont reliés au reste de l'ordinateur par ce qu'on appelle un **bus**. Ce bus est constitué de fils électriques dans lesquels "circulent" des signaux électriques (tensions ou courants) interprétés en 0 ou en 1. Ce bus est divisé en plusieurs sous-bus qui ont chacun une utilité particulière :

- Le **bus d'adresse** permet au processeur de sélectionner l'entrée, la sortie ou la portion de mémoire avec qui il veut échanger des données.
- Le **bus de commande** permet la gestion des échanges de données.
- Le **bus de donnée** par lequel s'échangent les informations.

Pour communiquer avec un périphérique, le processeur a juste besoin de configurer ces bus avec les bonnes valeurs. Mais communiquer avec un périphérique n'est pas aussi simple que ça, comme ce chapitre va vous le montrer.

Interfacage Entrées-sorties

Dans la façon la plus simple de procéder, le processeur se connecte au bus et va directement envoyer sur le bus : l'adresse, les données, et autres commandes à envoyer à l'entrée-sortie ou au périphérique. Ensuite, le processeur va devoir attendre et reste connecté au bus tant que le périphérique n'a pas traité sa demande correctement, que ce soit une lecture, ou une écriture.

Cette méthode a beau être simple, elle a un gros problème : nos périphériques sont très lents pour un processeur. Le processeur passe énormément de temps à attendre que le périphérique aie reçu ou envoyé sa donnée. Et il a donc fallu trouver des solutions plus ou moins élégantes.

Interfaçage

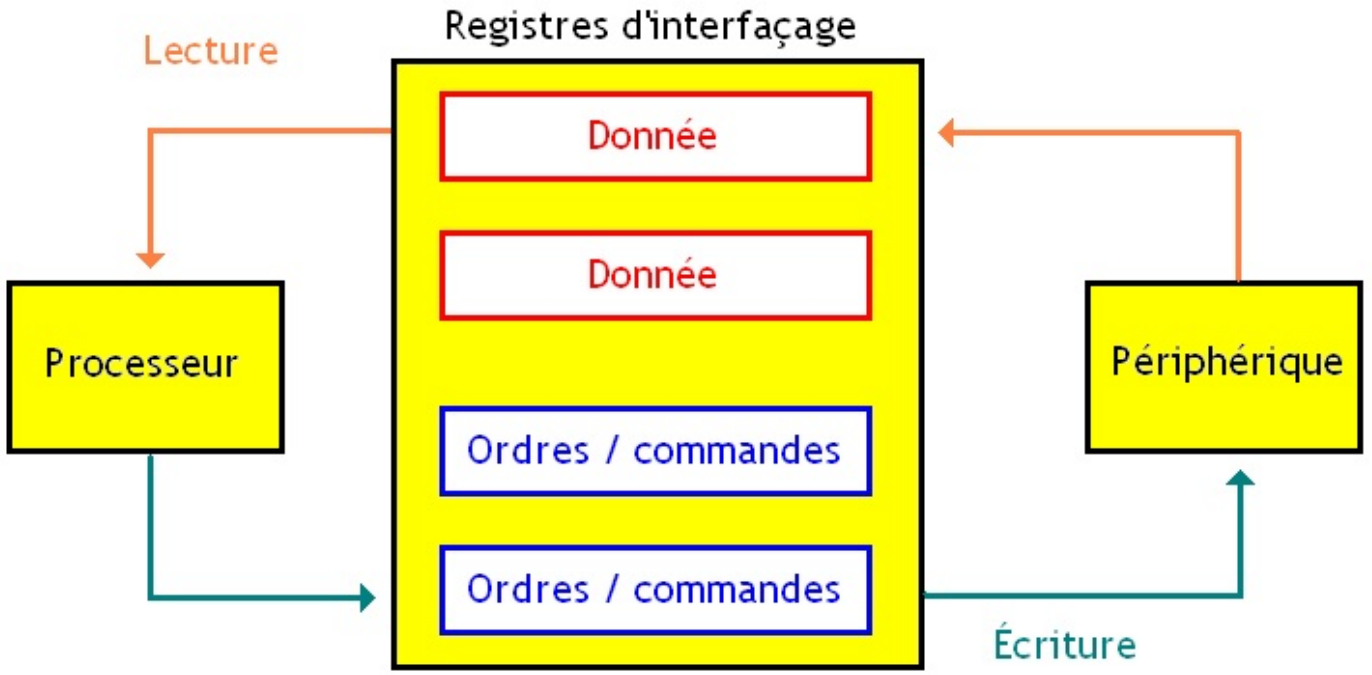
Pour faciliter la communication entre processeur et entrées-sorties/périphériques, une solution toute simple à été trouvée : **intercaler des registres entre le processeur et les entrées-sorties**. Ces registres servent à faciliter la communication avec le processeur : il suffit au processeur de lire ou écrire dans ces registres pour communiquer avec le périphérique .

Le processeur peut écrire des données sur une sortie sans monopoliser le bus en attendant que la sortie ou le périphérique aie reçu la donnée. Il écrit sa donnée dans les registres d'interfaçage et n'attend pas que le périphérique aie reçu la donnée. Cette sortie aura juste à lire le contenu du registre de façon régulière (ou quand elle aura détecté une écriture dans ce registre) pour voir si le processeur ou un autre composant lui a envoyé quelque chose.

Pour les entrées, la situation ne s'améliore pas vraiment pour le processeur : il doit continuer à lire le contenu des registres d'interfaçage régulièrement pour voir si un périphérique lui a envoyé quelque chose. C'est plus rapide que devoir scruter en quasi-permanence le bus : les processeur peut faire ce qu'il veut comme calcul, entre deux lecture du contenu du registre. Mais le problème demeure. Bien sûr, on va bientôt voir que diverses techniques peuvent permettre des améliorations parfois notables.

Registres d'interfaçage

Ces registres sont appelés des **registres d'interfaçage** et permettent au périphérique de communiquer avec "l'extérieur". C'est dans ces registres que le processeur va lire les informations que le périphérique veut lui transmettre, ou qu'il va écrire les données et ordres qu'il veut envoyer au périphérique.



Le contenu de ces registres dépend fortement du périphérique et on peut difficilement établir des règles générales quand à leur contenu.
Pour simplifier, on peut dire que ces registres peuvent contenir

- des **données** tout ce qu'il y a de plus normales ;
- ou des "**ordres**".

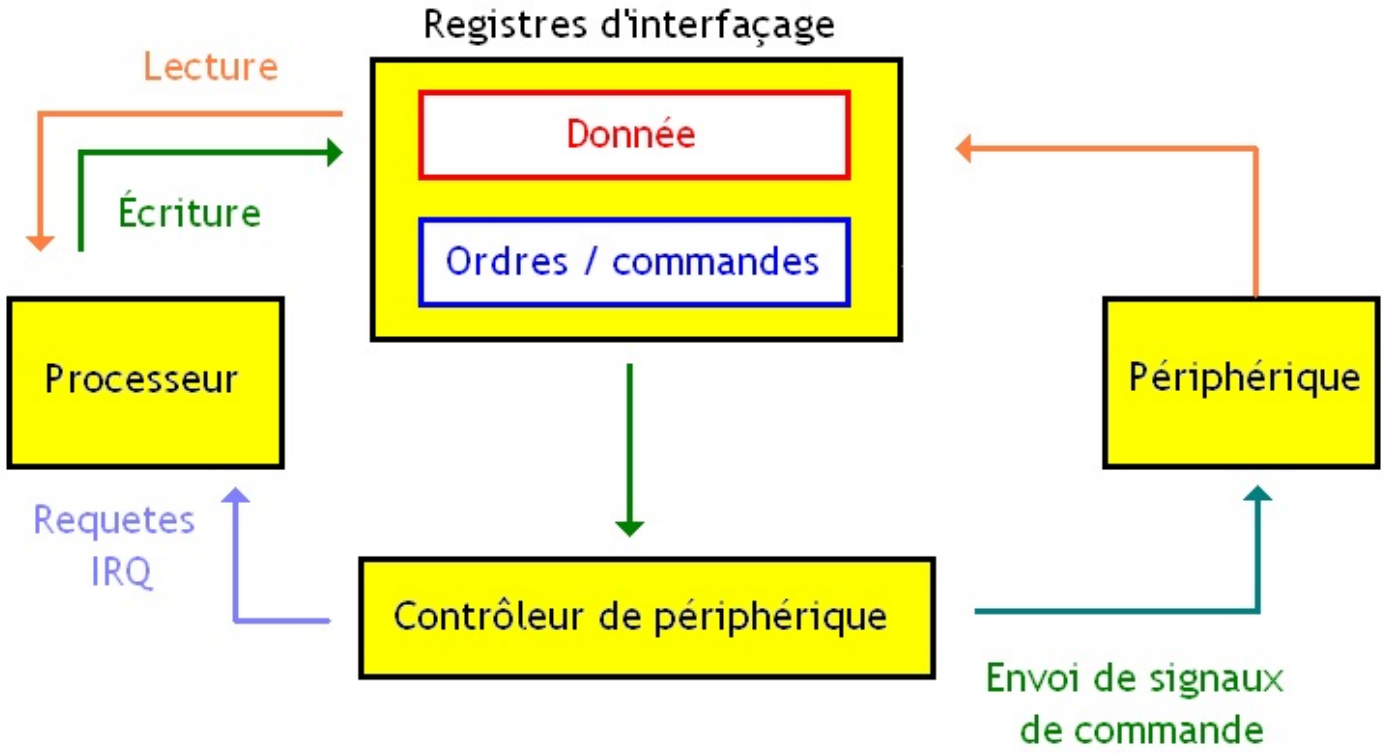
Ces ordres sont transmis au périphérique par des bits ou des groupes de bits individuels localisés dans des registres d'interfaçage. Le périphérique est conçu pour effectuer un ensemble d'actions préprogrammées. Suivant la valeur de ces bits ou groupes de bits, une de ces actions parmi toutes les autres sera sélectionnée, et ces bits ou groupes de bits peuvent donc servir à transmettre un "ordre". On peut comparer ces ordres avec les instructions d'un processeur, sauf qu'ici, ces ordres seront destinés à un périphérique.

Contrôleur de périphérique

Bon maintenant que notre processeur a écrit dans les registres d'interfaçage, qu'est-ce que se passe ?

A ce moment, un petit circuit nommé **contrôleur de périphérique** va alors prendre le relai. Celui-ci est un petit circuit électronique qui va lire les données et informations envoyées par le processeur, les interprète, et va piloter le périphérique de façon à ce que celui-ci fasse ce qui lui est demandé. Ce circuit peut être plus ou moins compliqué et dépend du bus utilisé, ainsi que du périphérique.

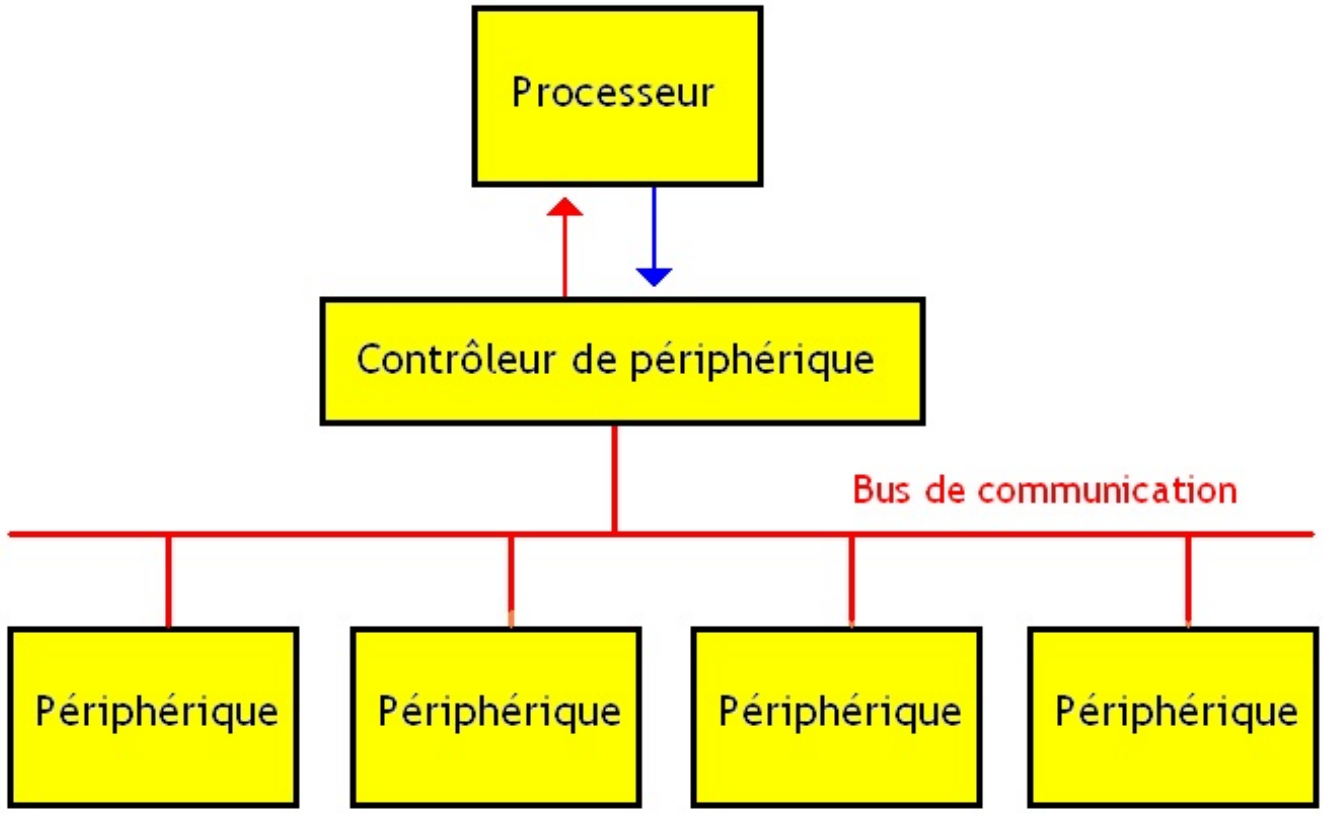
Vu que les ordres et les informations envoyées par le processeur sont stockés dans les différents registres reliés au bus, le contrôleur a juste à lire le contenu de ces registre et le traiter pour faire ce qui lui est demandé.



Comme vous le voyez, le boulot du contrôleur de périphérique est de générer des signaux de commande qui déclencheront une action effectuée par le périphérique. Ces signaux sont générés à partir du contenu des registres. L'analogie avec le séquenceur d'un processeur est possible, bien que limitée. Il faut bien remarquer qu'un contrôleur de périphérique travaille à la demande : on lui donne un ordre, il obéit et génère les signaux de commande.

Les contrôleurs de périphériques peuvent être très différents les uns des autres. Cela peut aller du simple circuit composé de quelques centaines de transistors à un petit micro-contrôleur, avec sa RAM, son CPU et son programme intégré. De plus, le contrôleur de périphérique peut très bien être séparé du périphérique qu'il va commander. Certains périphériques intègrent en leur sein ce contrôleur : les disques durs IDE, par exemple). Mais d'autres sont commandés par un contrôleur séparé du périphérique. Dans certains cas, ce contrôleur est placé sur la carte mère et peut même commander plusieurs périphériques en même temps : c'est le cas du contrôleur de bus USB.

Une précision assez importante s'impose : certains contrôleurs de périphériques peuvent permettre au processeur de communiquer avec plusieurs périphériques en même temps. C'est notamment le cas pour tout ce qui est contrôleurs PCI, USB et autres : ces contrôleurs sont reliés à un bus sur lequel plusieurs périphériques sont connectés. Le contrôleur se contente de prendre en charge l'échange d'informations via le bus en question et peut ainsi communiquer avec plusieurs périphériques.



Dans ce genre de cas, on peut parfaitement considérer que le contrôleur sert plus d'interface entre un bus spécialisé et le processeur.

Registre d'état

Certains de ces contrôleurs intègrent un registre qui contient des informations sur l'état du contrôleur, du périphérique ou du bus qui relie ces registres au périphérique. Ils peuvent être utiles pour signaler des erreurs de configuration ou des pannes touchant un périphérique.

Pilotes de périphériques

Lorsqu'un ordinateur utilise un système d'exploitation, celui-ci ne connait pas toujours le fonctionnement d'un périphérique et/ou de son contrôleur (par exemple, si le périphérique a été inventé après la création de l'OS). Il faut donc installer un petit programme qui va s'exécuter quand on souhaite communiquer avec le périphérique et qui s'occupera de tout ce qui est nécessaire pour le transfert des données, l'adressage du périphérique, etc. Ce petit programme est appelé **driver** ou **pilote de périphérique**.

La "programmation" d'un contrôleur de périphérique est très simple : il suffit de savoir quoi mettre dans les registres pour paramétrer le contrôleur. Et un pilote de périphérique ne fait que cela. Pour simplifier au maximum, un pilote de périphérique est un ensemble de petits sous-programmes qui ont chacun une utilité particulière. Chacun de ces sous-programmes s'exécute à la demande, quand un programme en a besoin. Ces sous-programme vont alors configurer les registres d'interfaçage de façon à ce que ceux-ci contiennent les ordres et données nécessaires pour que le contrôleur de périphérique fasse ce qu'on lui demande.

Problèmes

Avec les registres d'interfaçage, l'écriture est nettement plus rapide : le processeur écrit dans le registre adéquat et peut continuer son travail dans son coin en attendant que le périphérique aie fini. La seule contrainte, c'est que le processeur ne peut pas forcément (sauf cas particuliers) envoyer une autre commande au contrôleur de périphérique tant que la première commande n'est pas traitée. Tant que le contrôleur de périphérique n'est pas "libre", le processeur devra attendre.

Après avoir envoyé un ordre au contrôleur, le processeur ne sait pas quand le contrôleur redeviendra libre, et doit donc vérifier périodiquement si le contrôleur est prêt pour un nouvel envoi de commandes/données. Généralement, il suffit au processeur de lire le registre d'état du contrôleur : un bit spécial de celui-ci permet d'indiquer si le contrôleur est libre ou occupé. Pour la lecture, la situation est similaire : le processeur doit lire régulièrement son contenu pour voir si le périphérique ne lui a pas envoyé

quelque chose.

Cette méthode consistant à vérifier périodiquement si le périphérique a reçu ou envoyé quelques chose s'appelle le ***Pooling***. Cette technique permet de ne pas rester connecté en permanence durant le temps que met le périphérique pour effectuer une lecture ou une écriture, mais ce n'est pas parfait : ces vérifications périodiques sont autant de temps perdu pour le processeur. Pour solutionner ce problème, on a décidé d'utiliser des **interruptions** !

Interruptions

Encore un terme barbare ! Qu'est-ce que c'est qu'une interruption ?

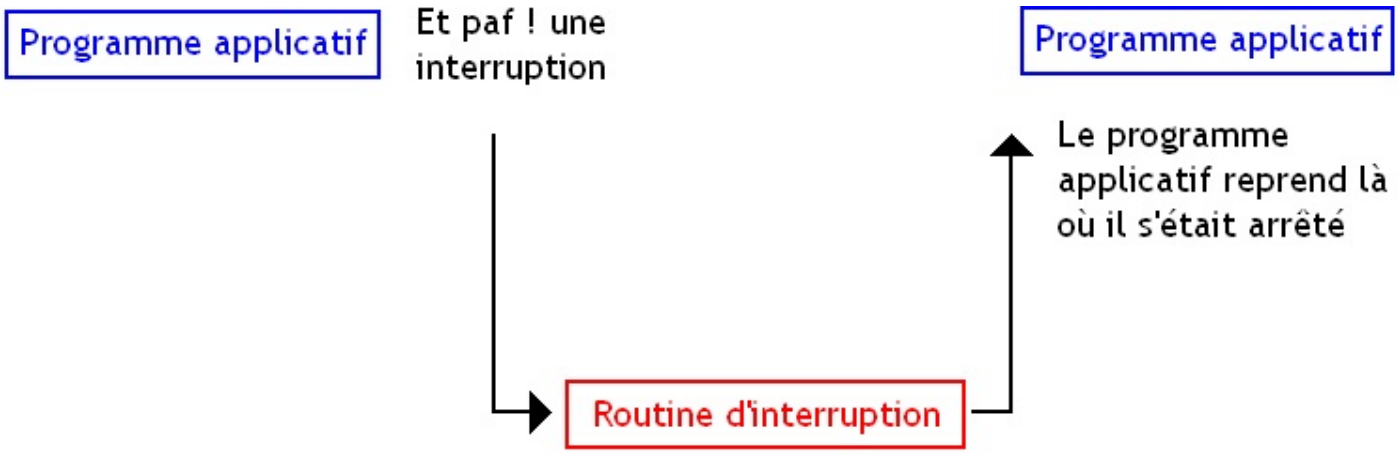
Ces interruptions sont des fonctionnalités du processeur qui vont interrompre temporairement l'exécution d'un programme afin de réagir à un événement extérieur (matériel, erreur fatale d'exécution d'un programme...) et de le traiter en temps voulu, avant de rendre la main au programme interrompu. Notre interruption va donc devoir effectuer un petit traitement (ici, communiquer avec un périphérique). Ce petit traitement est fait par un petit programme au auquel on a donné un nom technique : routine d'interruption.

On les utilise pour quelques cas bien précis, qui nécessitent un traitement ne pouvant attendre trop longtemps. Communiquer avec des périphériques est un de ces cas. Ainsi, pour communiquer avec une carte graphique, un disque dur ou encore avec le clavier, vous allez devoir utiliser des interruptions. Ces interruptions sont aussi utilisées pour permettre à plusieurs programmes de s'exécuter sur un processeur : on switche constamment d'un programme à un autre, à l'aide d'interruptions ; et pour bien d'autres choses encore. Il existe des processeurs qui ne gèrent pas les interruptions, mais ceux-ci sont vraiment très rare, vu leurs nombreuses utilisations diverses et variées.

Déroulement d'une interruption

Lorsqu'un processeur doit exécuter une interruption, celui-ci :

- Arrête l'exécution du programme en cours d'exécution et sauvegarde une partie ou l'intégralité de l'état du processeur (registres, cache, piles...)
- Exécute un sous-programme assez simpliste nommé **routine d'interruption**. (et donc, on doit effectuer une instruction de branchement vers ce sous-programme).
- Restaure l'état du programme sauvegardé afin de reprendre l'exécution de son programme là ou il en était.



En quoi ces interruptions permettent de communiquer efficacement avec des entrées-sorties ?

Très simplement : avec elles, le processeur n'a pas à vérifier périodiquement si le contrôleur de périphérique est libre ou a bien envoyé sa donnée. Les interruptions vont être utilisées pour prévenir que le contrôleur de périphérique a envoyé une donnée au processeur ou qu'il est libre. Ainsi, le processeur se contente d'écrire dans les registres d'interfaçage et ne fait rien de plus : on n'utilise le processeur que quand on en a besoin..

Prenons le cas d'une lecture ou un cas dans lequel le périphérique envoie une donnée au processeur (du genre : une touche a été tapée au clavier). Dans ce cas, le périphérique enverra une interruption au processeur pour le prévenir qu'une donnée lui est destinée. Le processeur traitera l'interruption en copiant le contenu des registres d'interfaçage contenant la donnée signalée par

l'interruption et fera ce qu'il faut avec.

Dans le cas d'une écriture ou d'un envoi de commande quelconque, le processeur se contentera d'écrire dans les registres d'interfaçage : le contrôleur préviendra qu'il est enfin prêt pour une nouvelle commande via une interruption.

Registres, again

Ce sous-programme va fatalement utiliser certains registres du processeur lors de son exécution. Comme pour les fonctions, il faut alors sauvegarder certains registres du processeur pour éviter que notre routine d'interruption vienne écraser des données qui ne doivent pas l'être. Ainsi, les registres utilisés par notre routine d'interruption seront sauvegardés au sommet de la pile de notre ordinateur.

Cette sauvegarde n'est pas toujours faite automatiquement par notre processeur. Parfois, c'est le programmeur qui doit coder lui-même la sauvegarde de ces registres dans la routine d'interruption elle-même. Il peut ainsi décider de ne pas sauvegarder certains registres qui ne sont pas destinés à être utilisés par la routine, afin de gagner un peu de temps.

Choix de la routine

Comme on l'a dit, une interruption a été conçue pour réagir à un événement, mais ce sont avant tout des programmes comme les autres, qui peuvent être exécutés comme n'importe quelle autre programme. Dans notre cas, ces interruptions seront simplement considérées comme des programmes simplistes permettant d'agir sur un périphérique. Bien sûr, devant la multiplicité des périphériques, on se doute bien qu'il n'existe pas d'interruption à tout faire : il va de soi qu'un programme envoyant un ordre au disque dur sera différent d'un programme agissant sur une carte graphique. Dans chaque cas, on aura besoin d'effectuer un traitement différent. On a donc besoin de plusieurs routines d'interruption.

Mais il faut bien décider quelle est l'interruption à exécuter suivant la situation. Par exemple, exécuter l'interruption de gestion du clavier alors qu'on souhaite communiquer avec notre disque dur donnerait un résultat plutôt comique. On va donc devoir stocker plusieurs de ces routines dans sa mémoire. Mais comment les retrouver ? Comme les autres données ! Chaque routine est donc placée dans la mémoire à un certain endroit, localisable par son adresse : elle indique sa position dans la mémoire.

Pour retrouver la position de notre routine en mémoire et savoir laquelle exécuter, certains ordinateurs utilisent une partie de leur mémoire pour stocker toutes les adresses de début de chaque routine d'interruption. En gros, cette partie de la mémoire contient toutes les adresses permettant de localiser chaque routine. Cette portion de la mémoire s'appelle le **vecteur d'interruption**. Pour chaque interruption, une partie fixe de la mémoire contiendra l'adresse de début de l'interruption à effectuer. Lorsqu'une interruption a lieu, le processeur va automatiquement aller chercher son adresse dans ce vecteur d'interruption.

Une autre solution est simplement de déléguer cette gestion du choix de l'interruption au système d'exploitation : l'OS devra alors traiter l'interruption tout seul. Dans ce cas, le processeur contient un registre qui stockera des bits qui permettront à l'OS de déterminer la cause de l'interruption : est-ce le disque dur qui fait des siennes, une erreur de calcul dans l'ALU, une touche appuyée sur le clavier, etc.

Priorité des interruptions

Et quand deux interruptions de déclenchent en même temps ?
Genre, quand le disque dur et le clavier souhaite informer le processeur qu'une lecture est finie et qu'il faut exécuter la routine correspondante ?

Et bien, dans ce cas là, on ne peut exécuter qu'une seule interruption. On doit donc choisir d'exécuter une interruption et pas l'autre. Le truc, c'est que certaines interruptions seront prioritaires sur les autres. Chaque interruption possède une **priorité**. Cette priorité est codée par un nombre : plus le nombre est élevé, plus l'interruption a une priorité faible ! Quand deux interruptions souhaitent s'exécuter en même temps, on choisit d'exécuter celle qui est la plus prioritaire (celle dont le nombre est le plus faible).

L'autre interruption n'est pas exécutée, et doit attendre. On dit que cette interruption est masquée. Le **masquage d'interruption** empêche l'exécution d'une interruption et la force à attendre un évènement précis pour pouvoir enfin s'exécuter. On peut néanmoins masquer des interruptions sans que cela soit du à l'exécution de deux interruptions simultanées, mais passons cela sous le tapis pour le moment.

Bien évidement, il faut décider quelles sont les interruptions les plus prioritaires sur les autres . Cela se fait par leur utilité : certaines interruptions sont plus urgentes que les autres. Une interruption qui gère l'horloge système est plus prioritaire qu'une interruption en provenance de périphériques lents comme le disque dur ou une clé USB.

Les différents types d'interruptions

Il y a trois moyens pour déclencher une interruption :

- une interruption déclenchée par une instruction de branchement un peu spéciale du processeur,
- les exceptions, qui se produisent automatiquement lorsque le processeur rencontre une erreur (du style une division par zéro),
- les requêtes d'interruptions, qui sont déclenchées par un événement d'origine matérielle.

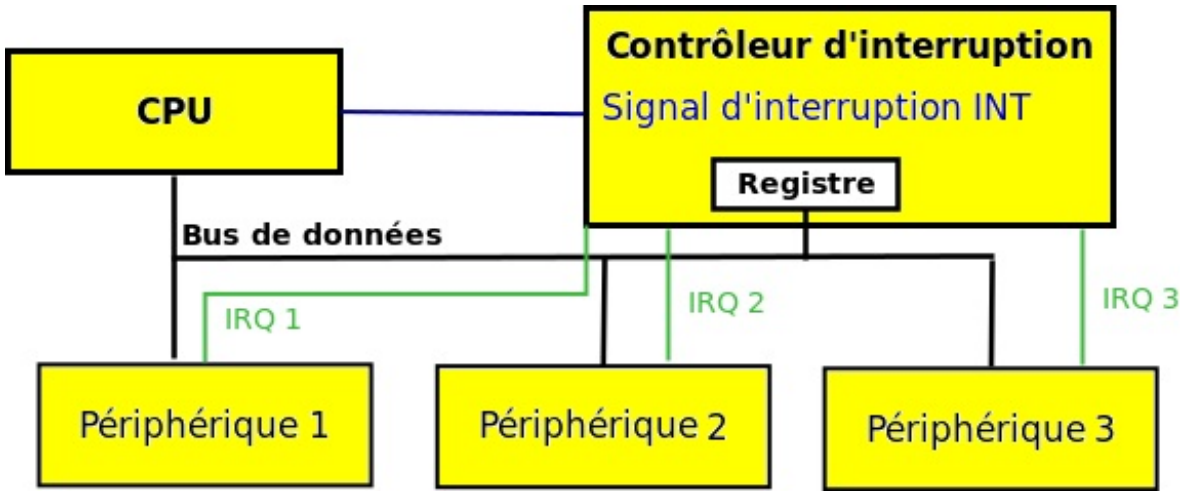
Comme vous le voyez, les interruptions peuvent non-seulement être appelées par un programme quelconque, grâce à l'instruction `int`, mais elles permettent aussi de réagir à des événements purement matériels, comme l'appui d'une touche au clavier.

IRQ

Les IRQ sont des interruptions qui sont déclenchées par un périphérique. Dans une implémentation simple des IRQ, notre périphérique est relié à notre processeur sur une de ses entrées. Si on envoie un 1 sur cette entrée, le processeur exécute une interruption parmi toute les autres. Ainsi, notre processeur contient autant d'entrée supplémentaires qu'il peut exécuter d'interruption.

Cela utilise un petit peu trop d'entrées qui pourraient être utilisées à autre chose. Pour éviter cela, on a inventé le contrôleur d'interruptions. Ce contrôleur d'interruptions est un petit circuit sur lequel on va connecter tous les fils d'IRQ. Ce contrôleur va recevoir sur ses entrées les IRQ envoyées par les périphériques.
Ce contrôleur possède :

- une sortie sur laquelle on envoie un signal sur une entrée du processeur qui va signaler qu'il faut exécuter une interruption ;
- un registre qui contient l'adresse de la routine sur laquelle il faut brancher ;
- et une "mémoire" ou un circuit qui contient (mémoire) ou déduit (circuit) pour chaque interruption, l'adresse de branchement de la routine à exécuter.



Le déroulement d'un interruption est alors très simple :

- une ou plusieurs signal d'IRQ arrivent en entrée du contrôleur d'interruptions.
- le contrôleur regarde si une des interruption doit être masquée ;
- il choisit la plus prioritaire ;
- il stocke dans son registre des informations permettant au processeur d'identifier le périphérique ou le contrôleur de périphérique qui a envoyé l'interruption ;
- il envoie un signal d'interruption au CPU ;
- le CPU lit le contenu du registre et en déduit quelle est la routine d'interruption à effectuer (son adresse dans le vecteur d'interruption par exemple) ;
- il exécute l'interruption.

Avec ce contrôleur, on évite au processeur de devoir gérer les priorités et les masquages, et on économise des entrées. Ce contrôleur est parfois placé à l'extérieur du processeur, et intégré dans le processeur pour les autres cas, comme l'on pouvait s'en douter.

Interruptions logicielles

Ces interruptions ont une cause différente : elle sont déclenchées par un programme en cours d'exécution sur notre ordinateur. Ces interruptions sont des instructions qui sont exécutables par un programme. Ainsi, le jeu d'instruction du CPU contient une instruction d'interruption. Un programmeur peut donc décider d'utiliser des interruptions à un certain moment de ce programme, pour des raisons particulières.

Par contraste, les IRQ ne sont pas des instructions appartenant au jeu d'instruction, et ne sont exécutées que quand une entrée-sortie en fait la demande. D'ailleurs, certains processeurs gèrent les IRQ, mais pas les interruptions logicielles.

Ces interruptions logicielles sont beaucoup utilisées par les pilotes de périphériques : ces interruptions logicielles peuvent être exécutées au besoin. Ainsi, les programmes d'un système d'exploitation utilisent des interruptions pour déléguer la communication avec les périphériques au pilotes de périphériques. Ces interruptions logicielles vont faire exécuter des routines chargées de lire ou écrire dans les registres d'interfaçage. Il va de soit que ces routines sont celles du pilote de périphérique. Ceux qui veulent en savoir plus peuvent aller lire mon tutoriel sur **les systèmes d'exploitation**.

Exceptions

Et maintenant, une petite digression, pour vous prouver que les interruptions peuvent servir à beaucoup de choses et pas seulement à communiquer avec des périphériques. On va parler des exceptions matérielle.

Ne pas confondre les exceptions matérielles et celles utilisées dans les langages de programmation, qui n'ont rien à voir !

Une **exception matérielle** est aussi une interruption, mais qui a pour raison un évènement interne au processeur, par exemple une erreur d'adressage, une division par zéro...Pour pouvoir exécuter des exceptions matérielles, notre processeur doit pouvoir déclencher une interruption lorsqu'une erreur particulière survient dans le traitement d'un instruction. Il faut donc que ce CPU intègre des circuits dédiés à cette tache.

Lorsqu'une exception matérielle survient, il faut trouver un moyen de corriger l'erreur qui a été la cause de l'exception matérielle : la routine exécutée va donc servir à corriger celle-ci. Bien sur, une exception matérielle peut avoir plusieurs causes. On a donc plusieurs routines.

Direct Memory Access

Avec nos interruptions, seul le processeur gère l'adressage de la mémoire. Impossible par exemple, de permettre à un périphérique d'adresser la mémoire RAM ou un autre périphérique. Il doit donc forcément passer par le processeur, et le monopoliser durant un temps assez long, au lieu de laisser notre CPU exécuter son programme tranquille. Pour éviter cela, on a inventé le ***bus mastering***. Grâce au *bus mastering*, le périphérique adresse la mémoire directement. Il est capable d'écrire ou lire des données directement sur les différents bus. Ainsi, un périphérique peut accéder à la mémoire, ou communiquer avec d'autres périphériques directement, sans passer par le processeur.

Arbitrage du bus

Le *bus mastering* n'est pas sans poser quelques petits problèmes : le processeur et/ou plusieurs périphérique peuvent bouloir accéder au bus en même temps. Manque de chance : on ne peut laisser deux composants tenter d'écrire des données en même temps sur le même bus : si on laissait faire ce genre de choses, on se retrouverait vite avec n'importe quoi sur notre bus !

Voici pourquoi : si on câble plusieurs composants sur le même bus, rien n'empêche ces deux composants de vouloir envoyer ou recevoir une donnée sur le bus en même temps. C'est ce qu'on appelle un **conflit d'accès au bus**. Cela pose problème si un composant cherche à envoyer un 1 et l'autre un 0 : le niveau logique du bit à envoyer est alors inconnu. Et quand on envoie plusieurs bits à la suite, tout ce que l'on reçoit à l'autre bout du fil est un espèce de mélange incohérent des deux données envoyées sur le bus par les deux composants. En clair : ça ne ressemble plus à rien à l'autre bout du fil !

Il faut donc trouver diverses méthodes pour gérer ces conflits d'accès, et choisir le plus équitablement et le plus efficacement possible quel composant peut envoyer ses données sur le bus. C'est ce qu'on appelle l'**arbitrage du bus**.

La technique du *bus mastering* est une technique assez générale, aussi je vais vous présenter, assez rapidement, une version de cette technique nommée le *direct memory acces*.

Direct Memory Acces

Le ***Direct Memory Acces*** est une technologie de *bus mastering* assez simple, qui permet à vos périphériques d'accéder à la mémoire RAM de votre ordinateur. Avec elle, le processeur n'est pas utilisé, ce qui rend la communication entre la mémoire et le

périphérique plus rapide. Elle peut même servir à transférer des données de la mémoire vers la mémoire, pour effectuer des copies de très grosses données, même si cela ne marche qu'avec du matériel particulier. Néanmoins, avec le *Direct Memory Acces*, le processeur doit tout de même intervenir au début et à la fin d'un transfert de données entre la mémoire et un périphérique.

Mais comment ça marche ?

Sans *Direct Memory Acces*, les périphériques et leurs contrôleurs ne peuvent pas modifier le contenu du bus d'adresse ou le bit R/W afin de demander une lecture ou une écriture : ils n'ont accès qu'aux registres d'interfaçage et peuvent déclencher des interruptions, mais le bus leur est interdit. Ainsi, les contrôleurs de périphériques ne peuvent adresser la mémoire et demander des opérations de lecture et/ou écriture directement : il doivent passer par un intermédiaire, à savoir notre bon vieux processeur.

Contrôleur DMA

Avec la technologie DMA, un circuit spécial souvent intégré à notre périphérique (ou à son contrôleur), le **contrôleur DMA** est relié au bus mémoire et peut modifier le contenu des bus d'adresse, de commande et de données. L'échange de donnée entre le périphérique et la mémoire est intégralement géré par celui-ci. Ce contrôleur DMA est similaire à un contrôleur de périphérique : il possède des registres dans lesquels le processeurs peut écrire et chacun de ces registres contient des données utiles au contrôleur pour faire ce qu'on lui demande.

Ce contrôleur DMA est capable de transférer un gros bloc de mémoire entre un périphérique et la mémoire. Le transfert peut aller dans les deux sens : du périphérique vers la RAM, ou de la RAM vers le périphérique. Le sens du transfert, ainsi que les informations sur le bloc de mémoire à transférer, sont précisés dans un des registres du contrôleur DMA.

Ce contrôleur incorpore donc des registres chargés de contenir :

- une adresse qui va définir le début du segment de la mémoire ;
- la longueur de ce segment de mémoire ;
- et parfois un ou plusieurs registres de contrôle.

Ces registres de contrôle peuvent contenir beaucoup de chose : avec quel périphérique doit-on échanger des données, les données sont-elles copiées du périphérique vers la RAM ou l'inverse, et bien d'autres choses encore.

Le travail du contrôleur est assez simple. Celui-ci doit se contenter de placer les bonnes valeurs sur les bus, pour effectuer le transfert. Il va donc initialiser le bus d'adresse à l'adresse du début du bloc de mémoire et initialiser le bus de commande selon la valeur du bit/registre spécifiant le sens de la transaction. Puis, à chaque fois qu'une donnée est lue ou écrite sur le périphérique, il va augmenter l'adresse de ce qu'il faut pour sélectionner le bloc de mémoire suivant.

Modes DMA

Il existe trois facon de transférer des données entre le périphérique et la mémoire. On peut ainsi citer le mode *block*, le mode *cycle stealing*, et le mode *transparent*. Ces trois modes différent par la façon dont le processeur est laissé libre de ses mouvements.

Dans le mode *block*, le contrôleur mémoire se réserve le bus mémoire, et effectue le transfert en une seule fois, sans interruptions. Cela a un désavantage : le processeur ne peut pas accéder à la mémoire durant toute la durée du transfert entre le périphérique et la mémoire. Alors certes, ça va plus vite que si on devait utiliser le processeur comme intermédiaire, mais bloquer ainsi le processeur durant le transfert peut diminuer les performances. Dans ce mode, la durée du transfert est la plus faible possible. Il est très utilisé pour charger un programme du disque dur dans la mémoire, par exemple. Et oui, quand vous démarrez un programme, c'est souvent un contrôleur DMA qui s'en charge !

Dans le mode *cycle stealing*, on est un peu moins strict : cette fois-ci, le contrôleur ne bloque pas le processeur durant toute la durée du transfert. En *cycle stealing*, le contrôleur va simplement transférer un byte (un octet) à la fois, avant de rendre la main au processeur. Puis, le contrôleur récupérera l'accès au bus après un certain temps. En gros, le contrôleur transfère un byte;, fait une pause d'une durée fixe, puis recommence, et ainsi de suite jusqu'à la fin du transfert.

Et enfin, on trouve le mode *transparent*, dans lequel le contrôleur DMA accède au bus mémoire uniquement quand le processeur ne l'utilise pas.

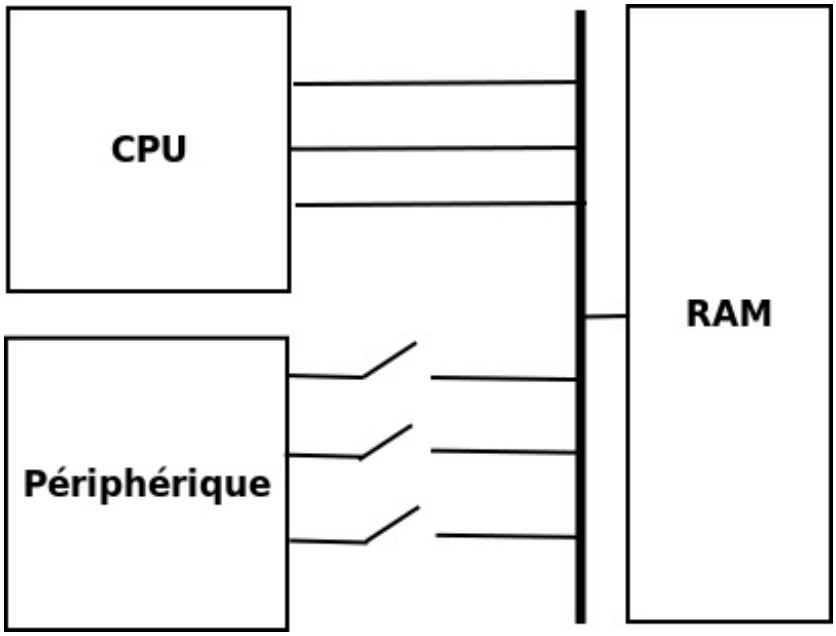
Un peu d'électricité

Premier bémol : le processeur et le contrôleur DMA sont tous deux reliés au bus. Et Cela pose problème ! Il faut arbitrer le bus et trouver éviter que le processeur et un périphérique envoient des données sur le bus en même temps.

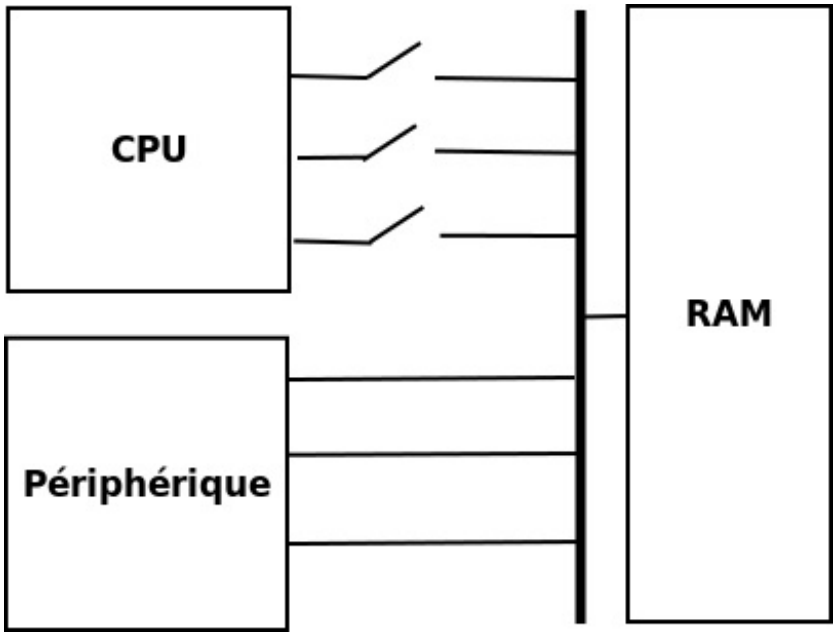
Les concepteurs de la technologie DMA sont des malins : ils ont trouvés une solution, basée sur les interruptions.
La solution retenue par le DMA est la suivante :

- le processeur et chaque périphérique est relié au bus par un transistor, un composant électronique qu'on utilisera en tant qu'interrupteur ;
- de plus, un fil relie directement le périphérique au bus, et donc à la mémoire (indirectement).

Tant que le périphérique n'accède pas à la mémoire, tout les interrupteurs reliant le périphérique au bus sont déconnectés : le périphérique n'est pas relié au bus. Par contre, le processeur est connecté au bus mémoire et peut faire ce qu'il veut dessus, du moment que c'est dans son programme.



Lorsqu'un périphérique souhaite accéder à la mémoire ou qu'un programme veut envoyer des données à un périphérique, il déclenche l'exécution d'une interruption. Pour qu'un périphérique puisse déclencher cette interruption, il suffit d'envoyer un bit sur le fil qui le relie au processeur. Dans le cas présent, la routine d'interruption va alors demander au processeur d'ouvrir ses interrupteurs ce qui le déconnecte du bus, tandis que le périphérique ferme ses interrupteurs. Le périphérique peut alors accéder à la mémoire.



Le contrôleur DMA s'occupera alors de l'échange de données, laissant le processeur libre d'exécuter ses calculs dans son coin, sans accéder au bus. A la fin de la transaction, le contrôleur DMA déconnecte le périphérique du bus. Pour prévenir le processeur de la fin d'un échange de donnée entre périphérique et mémoire, le contrôleur DMA enverra une interruption vers le

processeur qui permettra alors à celui-ci de se reconnecter au bus.

Adressage des périphériques

Le chapitre précédent a été assez instructif : maintenant, vous savez que nos périphériques sont reliés au processeur par l'intermédiaire d'un contrôleur de périphérique. Ce contrôleur de périphérique peut gérer un ou plusieurs périphériques en même temps, sans que cela ne pose problème. Mais il nous reste beaucoup de choses à expliquer, et certaines questions restent encore en suspend.

Comment faire pour accéder aux registres de ce contrôleur de périphérique ?

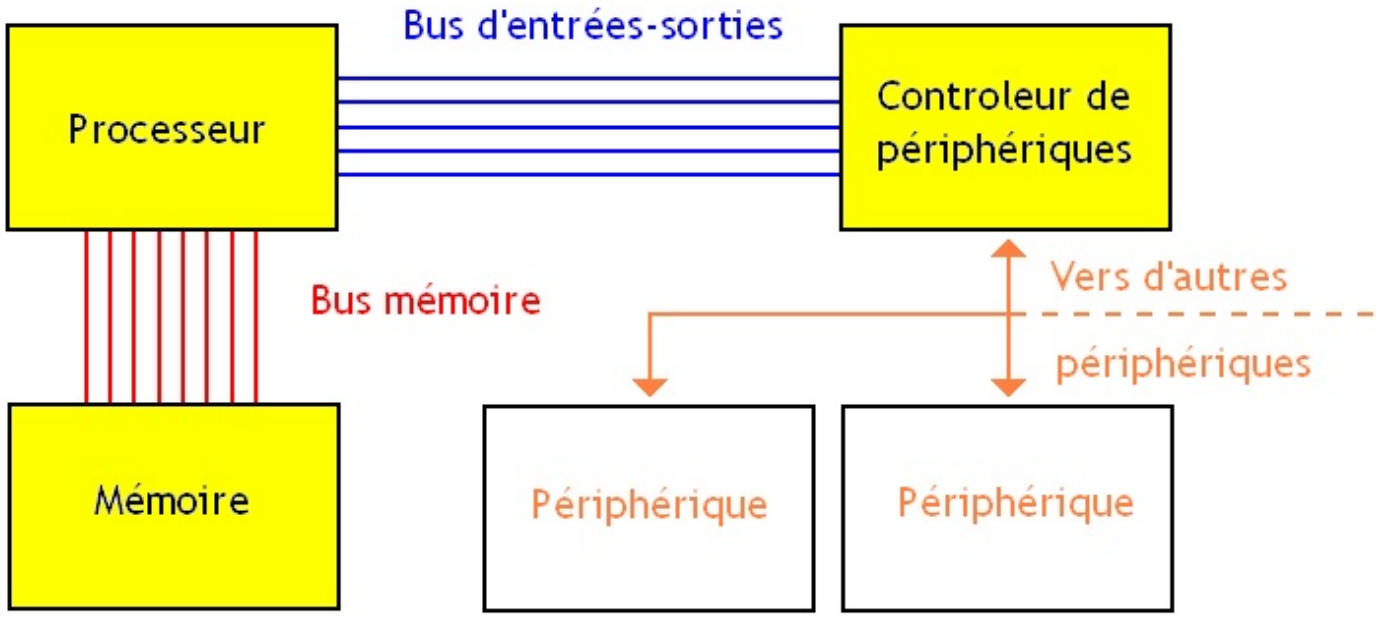
Ben oui, pour communiquer avec les registres du contrôleur, il doit bien exister un moyen pour pouvoir les localiser et les sélectionner ! Ce chapitre va vous montrer les différentes méthodes utilisées pour pouvoir "adresser" notre contrôleur de périphérique. Elles sont au nombre de trois et se nomment :

- la connexion directe ;
- les entrées-sorties mappées en mémoire ;
- et l'espace d'adressage séparé.

Connexion directe

Dans le cas le plus simple, le contrôleur est relié directement sur des entrées et des sorties du processeur : certaines entrées-sorties du processeur sont spécialement dédiées à la communication avec un périphérique ou un contrôleur de périphérique particulier. On se retrouve donc avec un bus supplémentaire, qui s'occupe de relier processeur et contrôleur de périphérique : le **bus d'entrées-sorties**.

Bus multiples



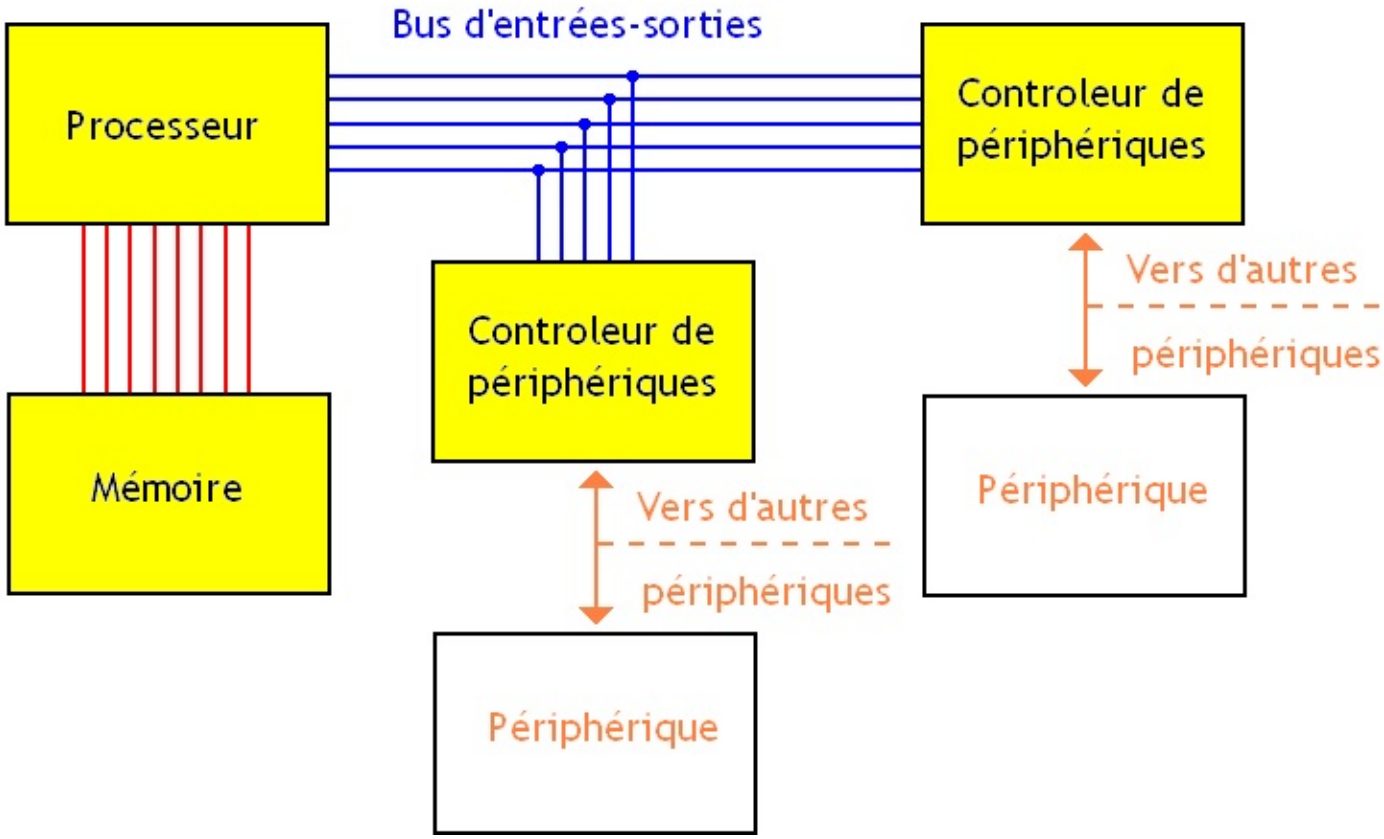
En faisant ainsi, on peut accéder à un contrôleur de périphérique et à la mémoire en même temps : on accède à la mémoire par le bus mémoire, et on accède à notre périphérique via le bus d'entrées-sorties. Dans ce genre de cas, le contrôleur n'a pas d'adresse qui permettrait de l'identifier. Le bus d'entrées-sorties se réduit donc à un bus de donnée couplé à un bus de commande.

Les problèmes commencent lorsqu'on se retrouve avec plusieurs contrôleurs de périphériques à câbler sur le processeur : on doit câbler autant de bus qu'on a des périphériques ! La quantité de fils utilisés, ainsi que le nombre de connexions à ajouter sur le processeur augmente beaucoup trop pour que ce soit possible. On doit donc trouver un moyen qui permette de gérer un grand nombre de périphériques et de contrôleurs qui soit viable techniquement parlant.

Bus d'entrées-sorties multiplexé

La première solution à ce problème est très simple : tous les contrôleurs de périphériques sont reliés au processeur par le même bus. On se retrouve donc avec deux bus : un spécialisé dans la communication avec les entrées-sorties, et un spécialisé dans la

communication avec la mémoire.



Avec cette solution, on doit trouver un moyen pour sélectionner le contrôleur de périphérique avec lequel on souhaite échanger des données. Pour cela, on est obligé d'utiliser l'adressage : chaque contrôleur de périphérique se voit attribuer une adresse, qui est utilisée pour l'identifier et communiquer avec lui. Notre bus d'entrées-sorties se voit donc ajouter un bus d'adresse en plus du bus de donnée et de commande.

En faisant ainsi, on peut accéder à un contrôleur de périphérique et à la mémoire en même temps : on accède à la mémoire par le bus mémoire, et on accède à notre périphérique via le bus d'entrées-sorties. Par contre, impossible d'accéder à plusieurs contrôleurs de périphériques en même temps : avec des bus séparés pour chaque contrôleur, on aurait éventuellement pu le faire, au prix d'un nombre de fils et de connexions impressionnant et très couteux.

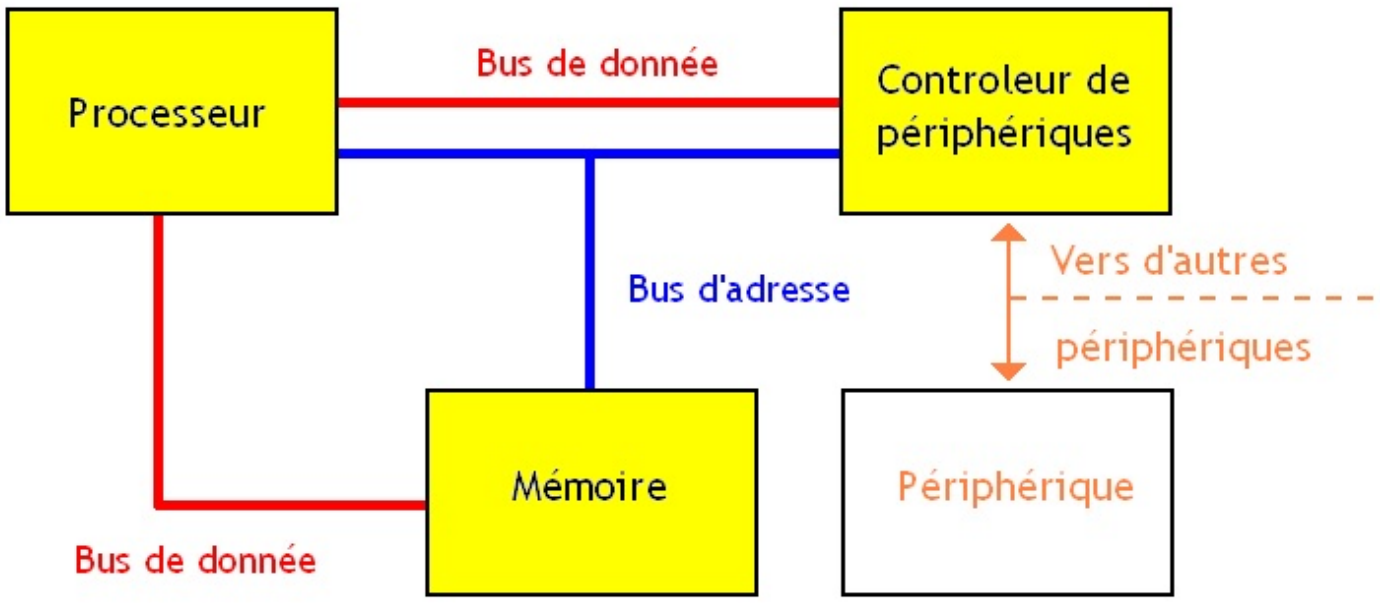
Il va de soit qu'avec cette solution, on économise beaucoup de fils : on n'a plus qu'un bus d'entrées-sorties à câbler sur le processeur, au lieu de devoir utiliser autant de bus d'entrées-sorties que de contrôleurs de périphériques. L'économie est énorme ! Sans compter que le processeur n'a pas à devoir gérer plusieurs bus d'entrée-sortie : il est moins complexe, ce qui fait gagner pas mal de transistors.

Espace d'adressage séparé

On l'a vu, avoir deux bus séparés pour la mémoire et les entrées-sorties est une bonne solution pour économiser des fils. Mais cela ne suffit malheureusement pas : on peut encore faire nettement mieux. En effet, on peut remarquer une chose assez évidente : le bus mémoire et le bus d'entrées-sorties possèdent chacun leur bus d'adresse.

Pourquoi ne pas mutualiser les deux ?

Et bien c'est une très bonne idée : on peut décider de partager les bus d'adresse du bus mémoire et du bus d'entrées-sorties sans problèmes. Ainsi, le même bus d'adresse sera utilisé pour les accès aux périphériques et pour les accès mémoires.

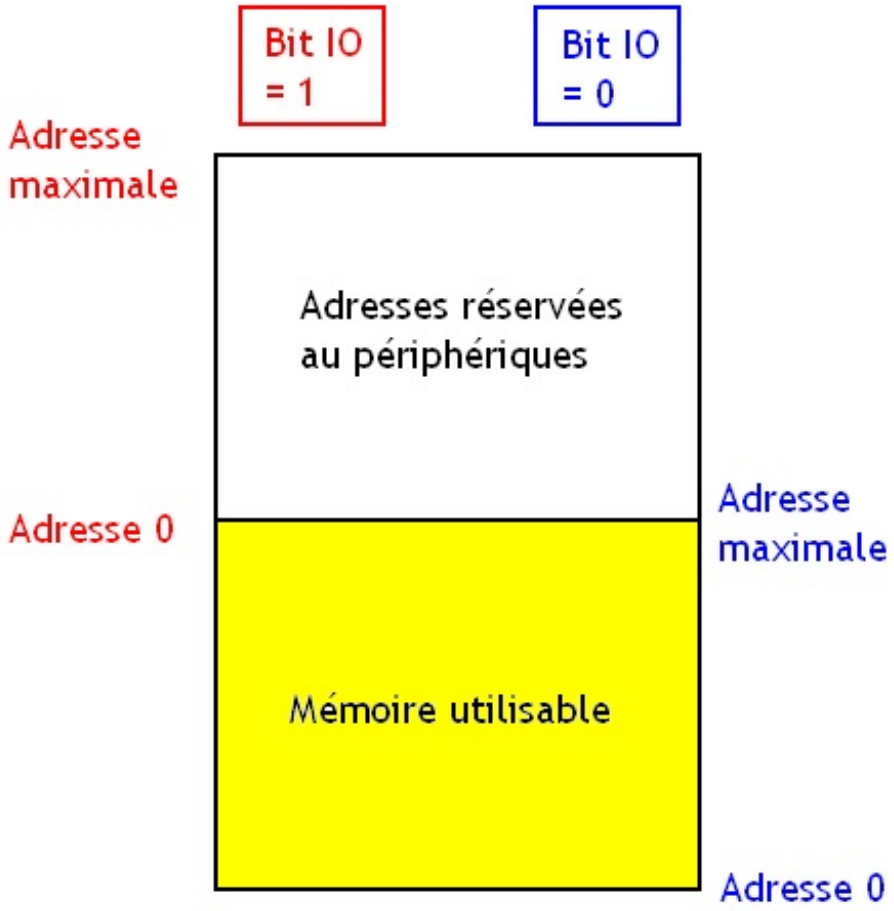


Par contre, les bus de données et de commande sont toujours séparés : on ne les mutualise pas !

Partage d'adresse

Et quand une case mémoire et un contrôleur de périphérique ont la même adresse, il se passe quoi ?

Aie ! Effectivement, cela arrive : on n'est pas à l'abri de ce genre de chose. Mais il y a une solution très simple : rajouter un fil sur le bus d'adresse qui servira à dire qu'on veut communiquer avec un contrôleur de périphérique, ou avec la mémoire. Ainsi, on rajoute un bit supplémentaire à notre adresse qui servira à distinguer la nature du composant avec lequel on veut communiquer : mémoire ou entrées-sorties. Ce bit sera appelé le **bit IO** dans la suite de ce tutoriel.



IO Instructions

Notre processeur est tout de même un peu perdu : comment faire pour que celui-ci positionne la valeur du bit IO à la bonne valeur et utilise le bon bus de donnée suivant qu'il veuille communiquer avec un périphérique ou avec la mémoire ?

Et bien on n'a pas vraiment le choix : on doit utiliser des instructions différentes suivant le composant avec lequel on communique. Ainsi, l'instruction qui écrira dans une adresse dédiée à un contrôleur de périphérique ne sera pas la même que celle chargée d'écrire dans une adresse en mémoire. Suivant l'instruction utilisée, le bit IO sera automatiquement positionné à la bonne valeur, et le bon bus de donnée sera sélectionné évitant toute confusion.

Cela a une conséquence : il est difficile de créer des instructions capables de transférer des données entre un périphérique et la mémoire directement. Généralement, les processeurs ne disposent pas d'instructions aussi spécialisées : cela demanderait beaucoup de circuits pour pas grand chose. Pour transférer des données d'un périphérique vers la mémoire, on est donc obligé de se servir d'un intermédiaire : les registres du processeur. Par exemple, prenons un transfert de donnée d'un périphérique vers la mémoire : on est obligé de copier la donnée du périphérique vers un registre du processeur, avant d'écrire le contenu de ce registre dans la case mémoire de destination. Évidement, cela est tout aussi vrai pour les transferts qui vont dans l'autre sens.

Pour éviter de devoir passer par un registre, on peut aussi utiliser la technique du *Direct memory acces*, qui résout totalement le problème. Mais il faut que notre périphérique ou notre carte mère incorpore un tel dispositif, ce qui n'est pas toujours le cas : ça peut couter cher ce genre de machin.

Entrées-sorties mappées en mémoire

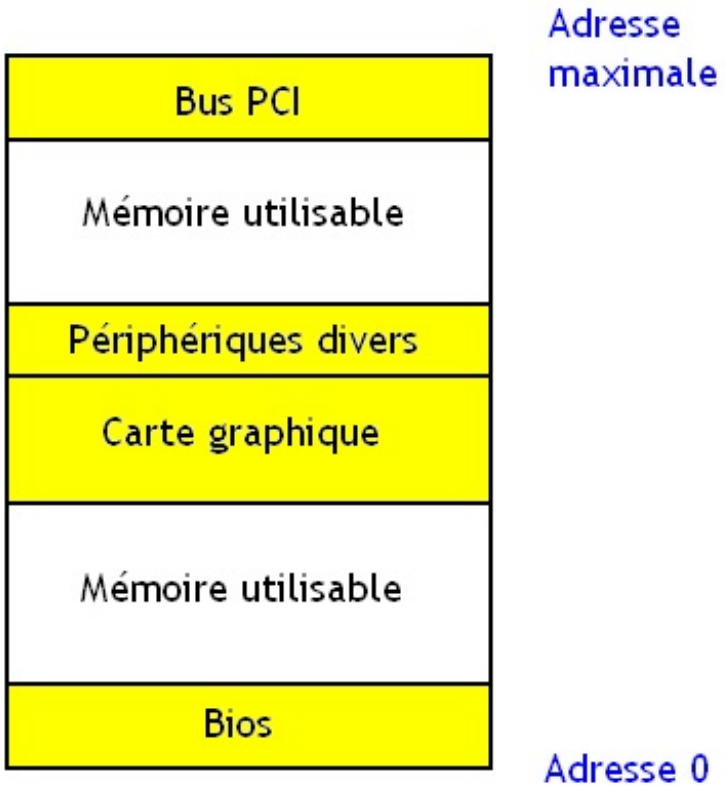
Mine de rien, on a réussit à économise pas mal de fils en partageant le bus d'adresse. Mais on se retrouve toujours avec un nombre de fils assez impressionnant à câbler. De plus, notre processeur doit disposer d'instructions de transfert de données différentes pour l'accès à la mémoire et l'accès aux périphériques : impossible de positionner le bit IO sans cette magouille. Mine de rien, partager le bus d'adresse complexifie pas mal la conception d'un processeur : doubler le nombre d'instructions d'accès mémoire impose de rajouter des circuits, ce qui n'est pas gratuit en terme de transistors et d'argent.

Pour éviter ce genre de désagrément, on a trouvé une autre solution : **mapper les entrées-sorties en mémoire**. Bon, cette phrase a l'air impressionnante, mais rassurez-vous : l'explication arrive.

Memory Mapped I/O

Le principe est simple : on n'utilise pas d'adresses séparées pour les périphériques (ou leurs contrôleurs) et la mémoire. Certaines adresses mémoires, censées identifier une case mémoire de la mémoire principale (la RAM), vont être redirigées vers les périphériques. En gros, tout se passe comme si le périphérique (ou son contrôleur) se retrouvait inclut dans l'ensemble des adresses utilisées pour manipuler la mémoire.

Bien sûr, le processeur n'a pas vraiment moyen de savoir qui est à l'autre bout du fil : il ne manipule que le contenu du bus, sans aucun moyen de savoir qui va recevoir la donnée.



Ainsi, certains "blocs" d'adresses mémoires seront invalidées en dur par des circuits électroniques chargés de gérer le bus, et renverront vers le périphérique sélectionné.

Perte de mémoire

On peut remarquer un petit défaut inhérent à cette technique : on ne peut plus adresser autant de mémoire qu'avant. Avant, toutes les adresses permettaient de sélectionner une case de la mémoire principale. Plus maintenant : certaines adresses mémoire sont réservées aux périphériques et ne peuvent plus être utilisées pour adresser la mémoire. Les cases mémoires en question deviennent inaccessibles.

Bon, ça peut paraitre théorique, et il faut bien avouer que l'on peut aisément penser que ça ne risque pas de nous concerner de sitôt. Et pourtant, ce problème touche nos ordinateurs modernes.

Vous avez déjà entendus parler du problème des 4 gigaoctets ?

C'est un problème souvent rencontré sur les forums : certaines personnes installent 4 gigaoctets de mémoire sur leur ordinateur et se retrouvent avec "seulement" 3.5-3.8 gigaoctets de mémoire. Ce "bug" apparait sur les processeurs x86 quand on utilise un système d'exploitation 32 bits. Avec ce genre de configuration, notre processeur utilise des adresses mémoires de 32 bits, ce qui fait 4gigaoctets de mémoire adressable.

Et bien parmi certaines de ces adresses, une partie est utilisée pour adresser nos périphériques et ne sert pas à adresser de la mémoire RAM : on perd un peu de mémoire. Et mine de rien, quand on a une carte graphique avec 512 mégaoctets de mémoire intégrée (cette mémoire est adressée directement et ca fait 512 Mo en moins d'un coup), une carte son, une carte réseau PCI, des

ports USB, un port parallèle, un port série, des bus PCI-Express ou AGP, et un BIOS à stocker dans une EEPROM FLASH, ça part assez vite.

Vous pouvez aussi tenter de regarder ce que ça donne pour d'autres machines qui ne sont pas des PC, vous trouverez exactement le même problème. Essayez de regardez sur le net ce que ça donne pour des ordinateurs tels que la Gamecube, les vielles consoles Atari, une Gameboy color, ou d'autres machines plus ou moins bizarres : cela vous montrera d'autres exemples assez intéressants.

Processeurs périphériques

Avec certains périphériques, cette technique qui consiste à mapper nos entrées-sorties en mémoire est poussée à l'extrême. C'est souvent le cas pour certains périphériques possédant une mémoire RAM intégrée dans leurs circuits : cartes graphiques, cartes sons, etc. Cette mémoire est directement accessible directement par le processeur en détournant suffisamment d'adresses mémoires. La mémoire de notre périphérique est accessible via des adresses mémoires normales : elle est ainsi en partie (voire totalement) partagée entre le processeur principal de l'ordinateur et le périphérique (et plus globalement avec tout périphérique pouvant adresser la mémoire).

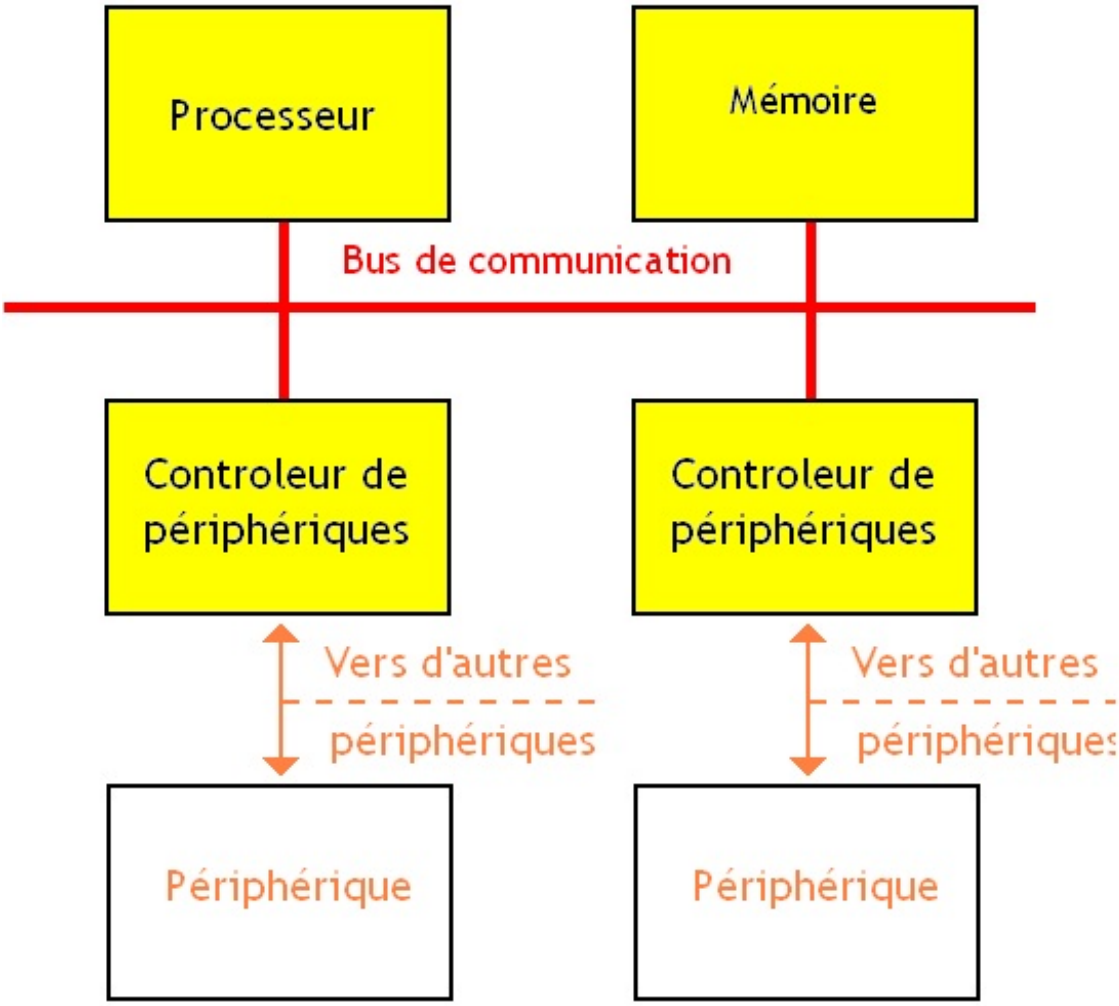
Certains de ces périphériques intégrant une mémoire RAM vont même plus loin et possèdent carrément un processeur pour des raisons de rapidité. Une grande partie de la gestion des entrées-sorties est ainsi déléguée au processeur intégré dans le périphérique et n'a pas à être gérée par le processeur principal de l'ordinateur.

Ce processeur intégré pourra ainsi exécuter des programmes chargés de commander ou configurer le périphérique. Ces programmes sont souvent fournis par les pilotes du périphérique ou le système d'exploitation. Ces programmes sont souvent recopiés par le processeur principal ou via un contrôleur DMA dans la mémoire du périphérique (partagée). Ainsi, adresser de tels périphériques pourra se faire assez directement en recopiant des programmes dans leur mémoire et en configurant quelques registres. Cela peut paraitre bizarre, mais sachez que c'est exactement ce qui se passe pour votre carte graphique si celle-ci a moins de vingts ans.

On peut considérer que ces périphériques sont des mini-ordinateurs, reliés à notre processeur.

Bus unique

Cette technique a un gros avantage : on peut utiliser un bus unique, sans rien dupliquer : fini le bus de donnée présent en double ! Avec cette méthode, on peut se contenter d'un seul bus qui relie le processeur aussi bien avec la mémoire qu'avec les périphériques.



Avec cette technique, on économise beaucoup de fils. Par contre, impossible d'accéder à la fois à la mémoire et à un contrôleur d'entrées-sorties : si le bus est utilisé par un périphérique, la mémoire ne pourra pas l'utiliser et devra attendre que le périphérique aie fini sa transaction (et vice-versa).

Et pour le CPU ?

Autre avantage : on n'a pas besoin d'instructions différentes pour accéder aux périphérique et à la mémoire. Tout peut être fait par une seule instruction : on n'a pas besoin de positionner un quelconque bit IO qui n'existe plus. Notre processeur possède donc un nombre plus limité d'instructions machines, et est donc plus simple à fabriquer.

Partie 6 : Hiérarchie mémoire

Vous savez depuis le chapitre sur la mémoire qu'il n'existe pas qu'une seule mémoire dans notre ordinateur. A la place, on utilise différentes mémoires, de tailles et de vitesses différentes, afin d'éviter d'utiliser une seule mémoire grosse et lente. Mais malgré cela, les mémoires sont aujourd'hui les composants qui limitent le plus les performances de nos programmes. De nombreuses optimisations existent pour faire en sorte que la lenteur de la mémoire ne soit pas un frein. Dans ce chapitre, nous allons voir lesquelles.

La mémoire virtuelle

De nombreuses instructions ou fonctionnalités d'un programme nécessitent pour fonctionner de connaitre ou de manipuler des adresses mémoires qui sont censées être fixes (accès à une variable, pointeurs, branchements camouflés dans des structures de contrôles, et d'autres). De nombreux modes d'adressages permettent de manipuler ou de calculer des adresses mémoires, plus ou moins efficacement.

Sur les ordinateurs qui n'exécutent qu'un seul programme à la fois, cela ne pose aucun problème : on sait exactement où va être chargé notre programme dans la mémoire. Mais sur des ordinateurs un peu plus compliqués, c'est rarement le cas : l'adresse à laquelle on va charger un programme dans notre mémoire RAM n'est presque jamais la même d'une exécution sur l'autre. Ainsi, les adresses de destination de nos branchements et les adresses de nos données ne sont jamais les mêmes. Il nous faut donc trouver un moyen pour faire en sorte que nos programmes puissent fonctionner avec des adresses qui changent d'une exécution à l'autre.

De plus, un autre problème se pose : un programme peut être exécuté sur des ordinateurs ayant des capacités mémoires diverses et variées et dans des conditions très différentes. Et il faut faire en sorte qu'un programme fonctionne sur des ordinateur ayant peu de mémoire sans poser problème. Après tout, quand on conçoit un programme, on ne sait pas toujours quelle sera la quantité mémoire que notre ordinateur contiendra, et encore moins comment celle-ci sera partagée entre nos différentes programmes en cours d'exécution : s'affranchir de limitations sur la quantité de mémoire disponible est un plus vraiment appréciable.

Et d'autres problèmes existent encore : nos processeurs ne gèrent pas la mémoire de la même façon. Si on devait prendre en compte cette gestion de la mémoire par le processeur lors de la conception de nos programmes, ceux-ci se seraient adaptés qu'à certains processeurs et pas à d'autres.

Solutions matérielles

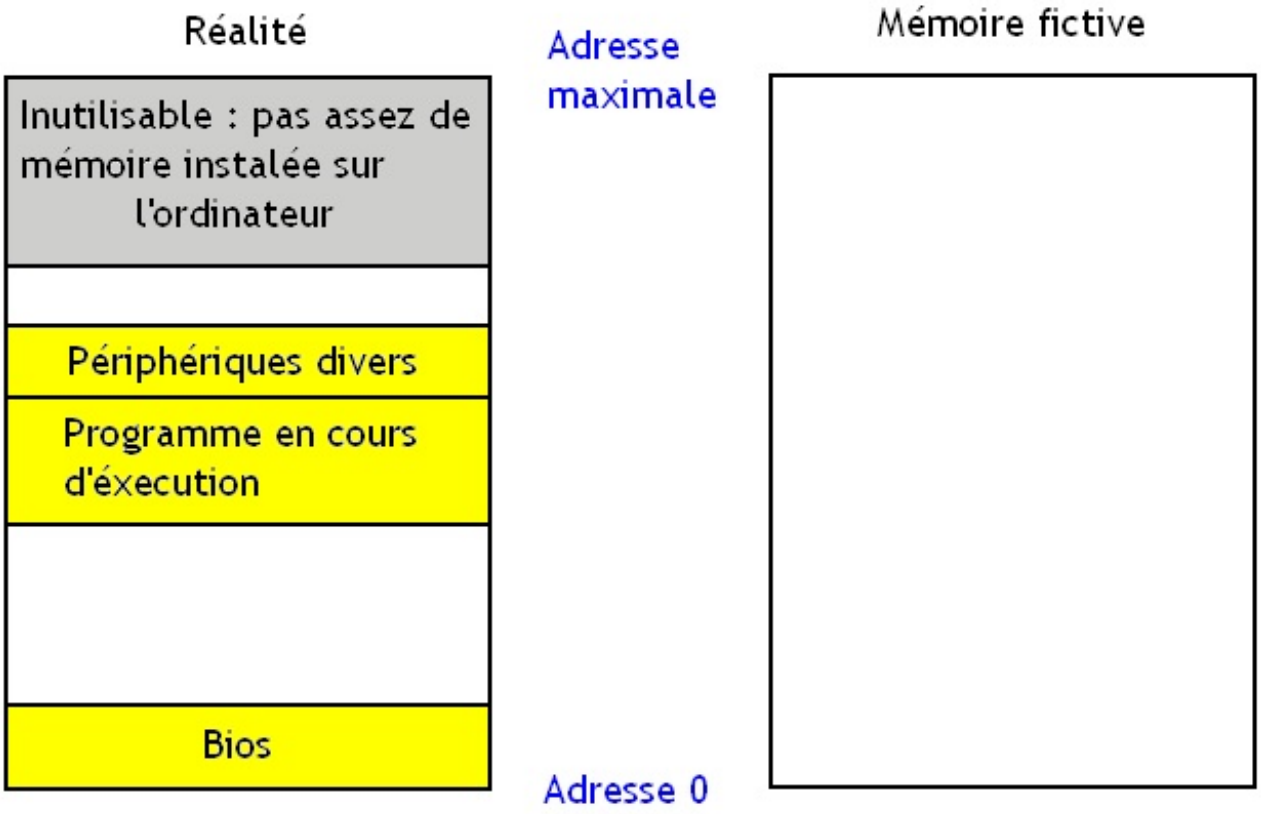
Vous l'avez compris, tous les détails concernant l'organisation et la gestion de la mémoire sont assez compliqué à gérer et faire en sorte que nos programmes n'aient pas à s'en soucier est un plus vraiment appréciable. Ce genre de problème a eu des solutions purement logicielles : on peut citer par exemple l'*overlaying*. C'est compliqué, long et donne des programmes utilisables sur un ordinateur en particulier et presque aucun autre : la compatibilité de ces programmes sur d'autres ordinateurs que celui pour lequel ils ont étés conçus est difficile voire impossible.

Mémoire virtuelle

Mais d'autres techniques règlent ces problèmes. Leur avantage : elles sont en partie voire totalement prises en charge par notre matériel, et c'est à celles-ci qu'on va s'intéresser. Ces solutions sont ce qu'on appelle la **mémoire virtuelle**. Le principe de la mémoire virtuelle est simple : donner aux programmes une représentation simplifiée de la mémoire qui contient plus de mémoire qu'installée sur notre ordinateur, et qu'un programme peut manipuler sans avoir à connaitre l'organisation exacte de la mémoire.

La fameuse représentation simplifiée de la mémoire dont on parle consiste simplement en une mémoire fictive, composée d'autant (voire plus) d'adresses que ce que le processeur peut adresser. Ainsi, on se moque des adresses inaccessibles, réservées aux périphériques, de la quantité de mémoire réellement installée sur l'ordinateur, ou de la mémoire prise par d'autres programmes en cours d'exécution.

Tout se passe comme si notre programme était seul au monde et pouvait lire et écrire à toutes les adresses disponibles à partir de l'adresse zéro. Bien sûr ces adresses sont des fausses adresses, des adresses fictives, mais c'est celles-ci qu'on retrouvera dans notre programme : toutes les adresses de destination de nos branchements, de nos données (modes d'adressages), et autres ; seront ces adresses fictives.



Vous remarquerez que notre programme a accès à plus d'adresses fictives que d'adresses réelles. Pour éviter que ce surplus de fausse mémoire pose problème, on utilise une partie des mémoires de masse (disque durs) d'un ordinateur en remplacement de la mémoire physique manquante.

Alors bien sûr, cette représentation fictive de la mémoire, cette mémoire virtuelle, n'est pas utilisable en l'état : on voit bien que notre programme va pouvoir utiliser des adresses auxquelles il n'a pas accès. Mais notre programme a malgré tout accès à une certaine quantité de mémoire dans la mémoire réelle.

Pour que cette fausse mémoire devienne utilisable, on va transformer les fausses adresses de cette fausse mémoire en adresses réellement utilisables. Ces fausses adresses sont ce qu'on appelle des **adresses logiques** : ce seront les adresses manipulées par notre programme, grâce aux divers modes d'adressages vus plus haut, et qu'on retrouvera dans nos registres. Par contre, les adresses réelles seront ce qu'on appelle des **adresses physiques**.

La MMU

Maintenant, il ne nous reste plus qu'à implémenter cette technique.

Mais comment faire ?

Comme à chaque fois qu'on a un problème à résoudre : on rajoute un circuit.

Ici, le circuit qu'on va rajouter va se charger de traduire automatiquement les adresses logiques manipulées au niveau du processeur en adresses physiques qui seront envoyées sur le bus d'adresse. Un tel circuit sera appelé la **Memory Management Unit**. Un des rôles qu'elle va jouer dans notre ordinateur sera de faire la traduction adresses logiques -> adresses physiques. Et j'ai bien dit un des rôles : elle en a d'autres, plus ou moins complexes.

Il faut préciser qu'il existe différentes méthodes pour gérer ces adresses logiques et les transformer en adresses physiques : les principales sont la segmentation et la pagination. La suite du tutoriel va détailler ces deux techniques, et quelques autres.

Segmentation

La première technique de mémoire virtuelle qu'on va aborder s'appelle la **segmentation**. Cette technique consiste à découper notre mémoire virtuelle en gros blocs de mémoire qu'on appelle **segments**.

Généralement, on ne découpe pas ces blocs n'importe comment : on préfère que ce découpage soit logique et reflète un peu l'organisation du programme qu'on est en train de découper en segments. Par exemple, il est assez courant de découper nos programmes en plusieurs zones bien distinctes :

Segment	Utilité
Text	Il sert à stocker la suite d'instructions qui consiste notre programme : il s'agit d'un segment qui rassemble tout le contenu de la mémoire programme (souvenez-vous des premiers chapitres de la partie 2). Ce segment a presque toujours une taille fixe.
Data	Le segment *data* contient des données qui occupent de la mémoire de façon permanente. Ce segment *Data* est un segment dans laquelle on stocke des données définitivement. Impossible de libérer la mémoire de ce segment pour faire de la place. En conséquence, ce segment a toujours une taille fixe.
Stack	Il s'agit de notre pile : celle-ci possède un segment rien que pour elle.
Heap	Le *Heap*, ou tas, est un segment qui sert à stocker des données, un peu comme le segment *Data*. Mais à l'inverse de ce dernier, on peut effacer les données qu'il contient lorsqu'elles deviennent inutiles. Ce tas a donc une taille variable.

Grosso modo, on retrouve cette organisation dans beaucoup de programmes conçus pour utiliser la segmentation, à quelques détails prêt. Par exemple, on peut trouver d'autres segments supplémentaires plus ou moins utiles comme des segments d'état de tâche, mais cela nous entrainerait trop loin. Comme vous le voyez, ce découpage est assez cohérent avec l'organisation d'un programme en mémoire.

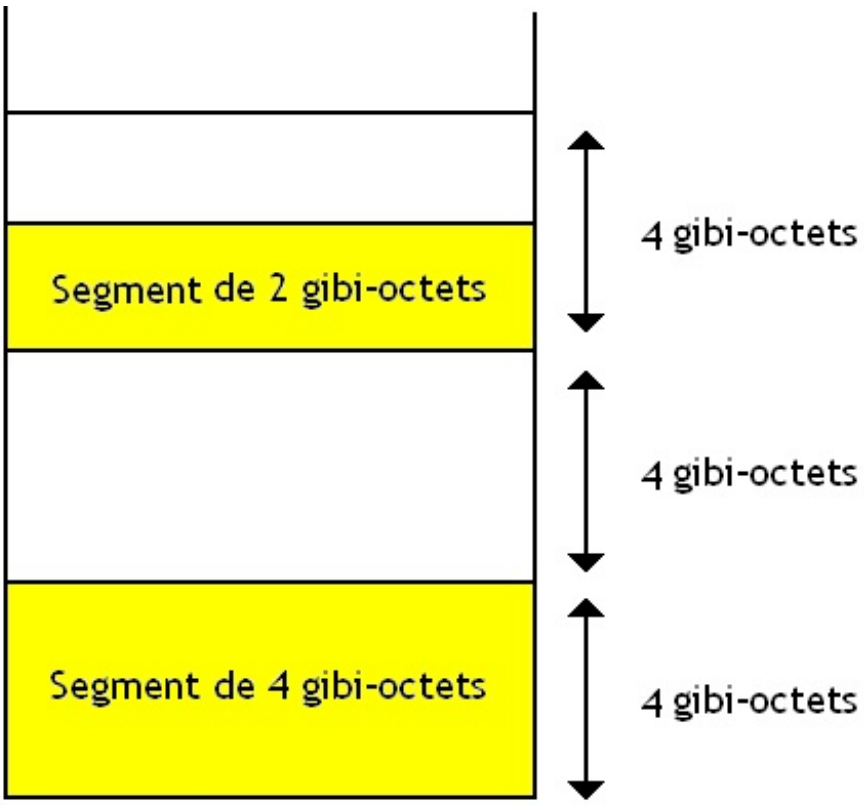
Principe

Pour donner un aperçu de la technique de segmentation, je vais prendre un petit exemple assez simple : la technique de segmentation telle qu'elle est utilisée sur les processeurs x86.

Segment

Sur ces processeurs x86, la mémoire virtuelle est bien plus grande que l'espace d'adressage du processeur. Sur un processeur 32 bits, capable d'adresser 2^{32} *bytes* de mémoire, cette mémoire virtuelle utilise des adresses de 48 bits. Cette mémoire virtuelle peut être découpée en 2^{16} segments de taille variable, pouvant faire jusqu'à maximum 4 gibi-octets.

Chaque segment commence donc à une adresse bien particulière : cette adresse est un multiple de 4 gibi. En clair, les 32 bits de poids faible de l'adresse de début d'un segment valent tous 0. En clair, tout se passe comme si un segment avait droit à un espace mémoire de 2^{32} bits rien que pour lui, qu'il peut utiliser à loisir : on en revient à notre description de départ de la mémoire virtuelle.



Adresse virtuelle

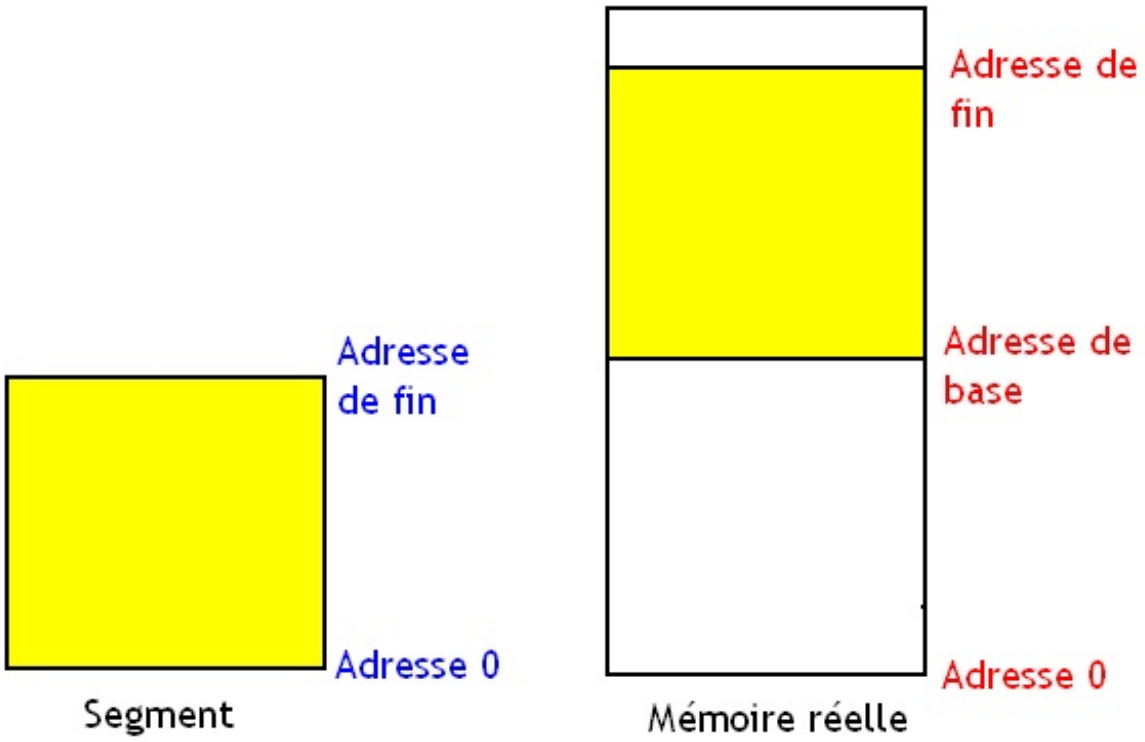
Toute donnée localisée dans le segment peut donc être localisée à partir de l'adresse de base du segment : si on connait le segment, et la position de la donnée dans le segment, on peut en déduire l'adresse finale de notre donnée. Pour identifier un segment, seuls les 16 bits de poids forts de l'adresse 48 bits sont utiles. Ceux-ci permettent de préciser le segment dont on parle : on les appelle le **sélecteur de segment**. A ce descripteur de segment, on rajoute donc un **offset**, qui permet de déterminer la position de notre donnée dans le segment.

Voici donc à quoi ressemble une adresse logique 48 bits :

Sélecteur	*Offset*
16 bits	32 bits

Relocation

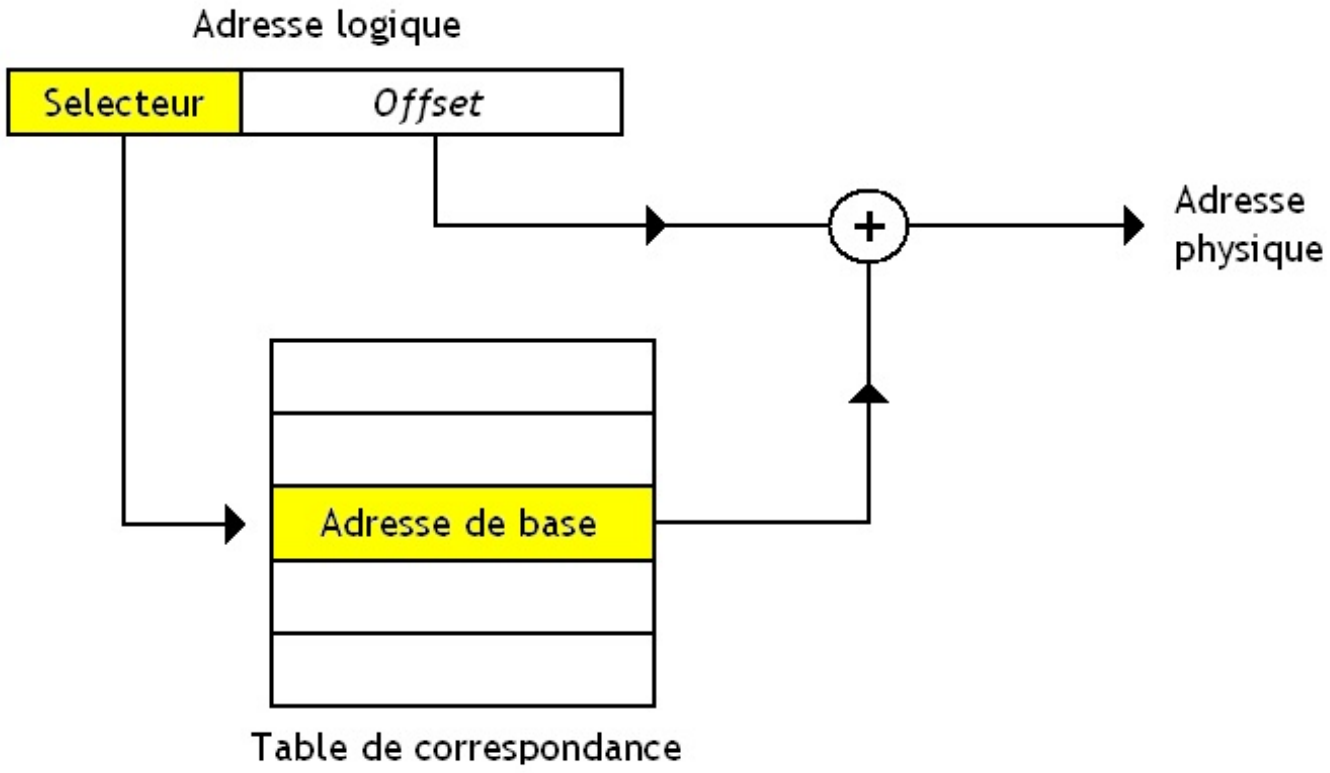
Chacun de ces segments peut être placé n'importe où en mémoire physique. C'est le premier avantage de la segmentation : un segment n'a pas d'adresse bien fixée en mémoire physique et peut être déplacé comme bon nous semble. Mais par contre, l'organisation des données dans un segment n'est pas modifiée.



Généralement, lorsqu'on veut charger un segment en mémoire à partir du disque dur, celui-ci sera placé quelque part en mémoire RAM, à une certaine adresse. C'est le système d'exploitation qui gère le placement des segments dans la mémoire physique.

Calcul de l'adresse physique

Pur calculer l'adresse d'une donnée en mémoire physique, il nous suffit donc d'ajouter l'adresse de base de notre segment en mémoire physique, et l'*offset* (qui est, rappelons-le, la position de notre donnée dans le segment). Il nous faut donc se souvenir pour chaque segment de son adresse de base, l'adresse à laquelle il commence. Cette correspondance segment-adresse de base est stockée soit dans une table de correspondance en mémoire RAM, soit dans des registres du processeur.



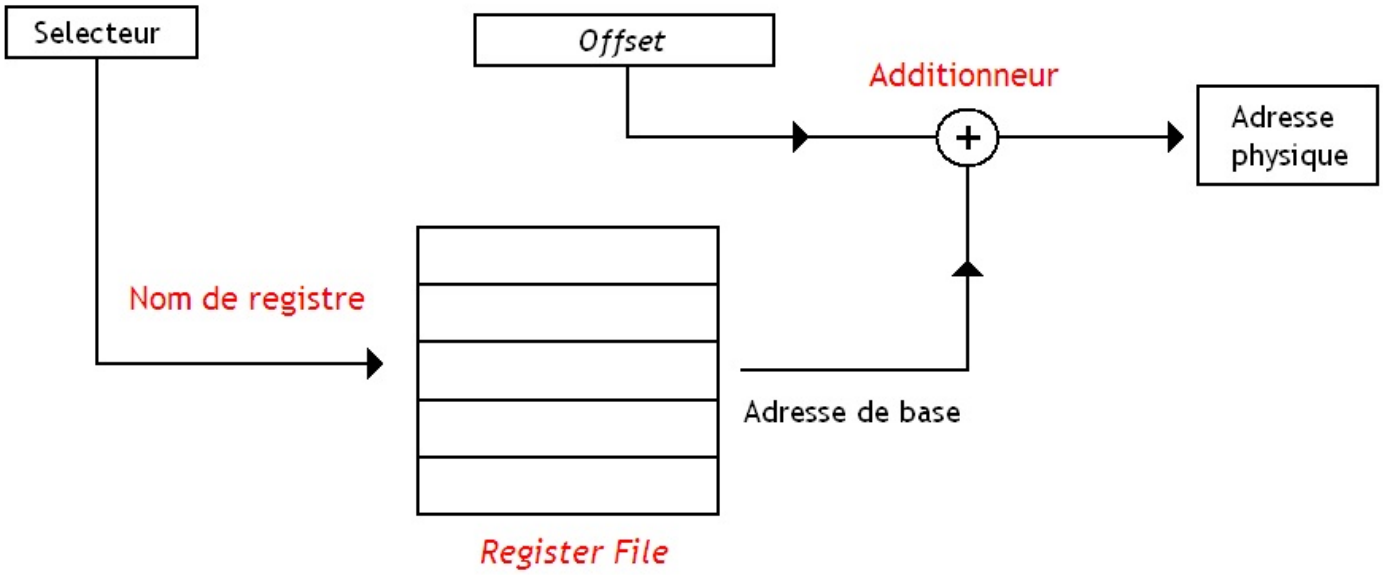
Il faut préciser que cette table de correspondance est unique pour chaque programme : chaque programme manipule les mêmes adresses virtuelles. Ce qui fait que deux programmes peuvent manipuler des adresses logiques identiques dans leur mémoire fictive qui leur est attribuée. Pourtant, les données correspondant à ces adresses logiques seront différentes et seront stockées dans des adresses mémoires physiques différentes. Pour éviter les problèmes, on n'a pas le choix : il faut soit utiliser des tables de correspondances différentes pour chaque programme.

MMU

La MMU d'un processeur implémentant la segmentation est donc assez simple. Il s'agit d'un circuit qui prend en entrée une adresse virtuelle et renvoie en sortie une adresse physique. On a vu que pour faire cette traduction adresse virtuelle -> adresse physique, notre MMU a besoin de savoir faire une addition, ainsi que d'une table de correspondance. Notre MMU contient donc un additionneur pour faire l'addition entre l'adresse de base et l'*Offset*. Reste à gérer le cas de la table de correspondance.

Dans le cas le plus simple, cette table de correspondance est stockée en mémoire RAM. Et donc, pour chaque accès à une adresse virtuelle, notre processeur va non seulement accéder à l'adresse physique, mais il va aussi devoir effectuer une lecture en RAM pour récupérer l'adresse de base du segment.

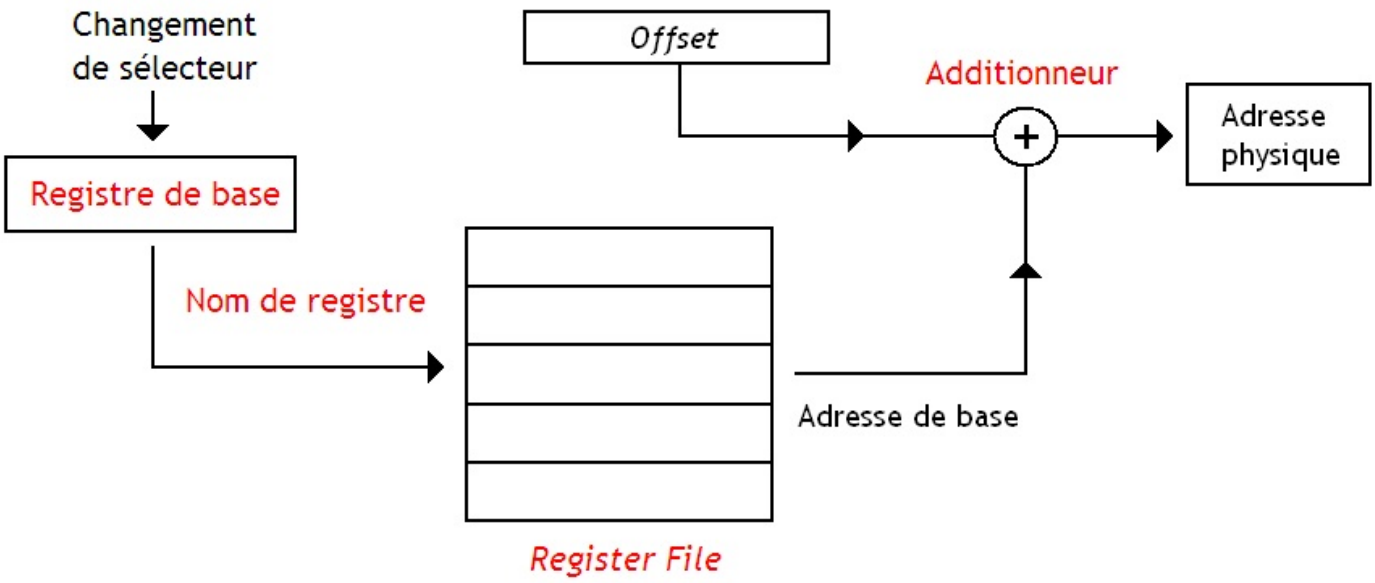
Ceci dit, devoir accéder à la RAM à chaque accès mémoire est une perte de temps. Pour éviter d'avoir à accéder à une table de correspondance stockée en mémoire, on peut stocker celle-ci dans un *Register File*. Le sélecteur de segment peut donc être considéré comme un nom de registre qui va préciser dans quel registre est placé l'adresse de base du segment à manipuler.



Comme je l'ai dit plus haut, notre processeur utilise des tables de correspondances différentes pour chaque programme. Lorsque notre processeur veut changer de programme, il doit pouvoir charger le contenu de cette table de correspondance dans les registres. Ce chargement est souvent effectué par le système d'exploitation.

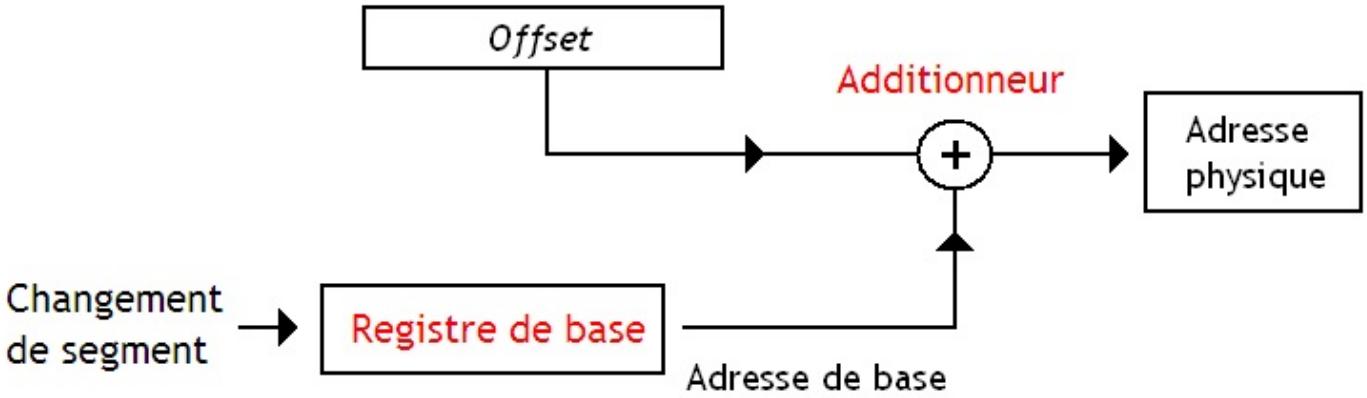
Mais il y a moyen de faire plus simple. Si on remarque bien, on n'est pas obligé de garder les adresses de base de tous les segments dans nos registres. Dans la grosse majorité des cas, il est rare que notre processeur doive accéder à deux segments simultanément. Et les changements de segments sont rares. Quand on accède à un segment, on peut être sur que notre processeur va effectuer un grand nombre d'accès dans ce segment avant d'en changer.

Dans ces conditions, indiquer le sélecteur dans chaque adresse mémoire serait une catastrophe : imaginez un peu l'impact sur la quantité de mémoire prise par notre programme ! A la place, il est possible de préciser le sélecteur une fois pour toute, via une instruction spéciale. Ce sélecteur est alors mémorisé dans un registre, le **registre de base**. Tous les accès mémoire suivants utiliseront ce sélecteur, et n'auront pas à le préciser : on évite d'avoir à préciser le sélecteur à chaque fois.



Pour implémenter cette technique, nous avons juste besoin de rajouter un registre pour stocker ce sélecteur, et quelques instructions qui permettent de modifier ce registres : ces instructions serviront à changer de sélecteur.

On peut encore améliorer la situation en faisant en sorte que notre registre de base ne stocke pas le sélecteur du segment, mais directement son adresse de base. Notre MMU est alors un peu plus rapide.



Dans ce cas, la gestion du contenu de ce registre de base est déléguée au système d'exploitation ou à certains circuits du processeur.

Protection mémoire

La segmentation nous permet donc de relocaliser nos programmes où l'on veut en mémoire RAM. Mais la segmentation ne permet pas que cela : elle permet aussi d'interdire certaines manipulations dangereuses sur la mémoire.

Généralement, plusieurs programmes sont présents en même temps dans notre ordinateur. Bien sur, on ne peut, sur une machine à un seul processeur, exécuter plusieurs programmes en même temps. Mais les autres programmes démarrés par l'utilisateur ou par le système d'exploitation, doivent absolument se partager la mémoire RAM, même s'il ne s'exécutent pas. La cohabitation de plusieurs programmes pose en effet quelques problèmes.

Si un programme pouvait modifier les données d'un autre programme, on se retrouverait rapidement avec une situation non-prévue par le programmeur. Cela a des conséquences qui vont de comiques à catastrophiques, et cela fini très souvent par un joli plantage.

Comment éviter qu'un programme accède à une donnée d'un autre programme ?

Très simple : on définit pour chaque programme des portions de la mémoire dans laquelle il pourra écrire ou lire. Le reste de la mémoire sera inaccessible en lecture et en écriture, à part quelques petites parties de la mémoire, partagées entre différents programmes. Aucun autre programme ne pourra alors lire ou écrire dans cette partie réservée de la mémoire.

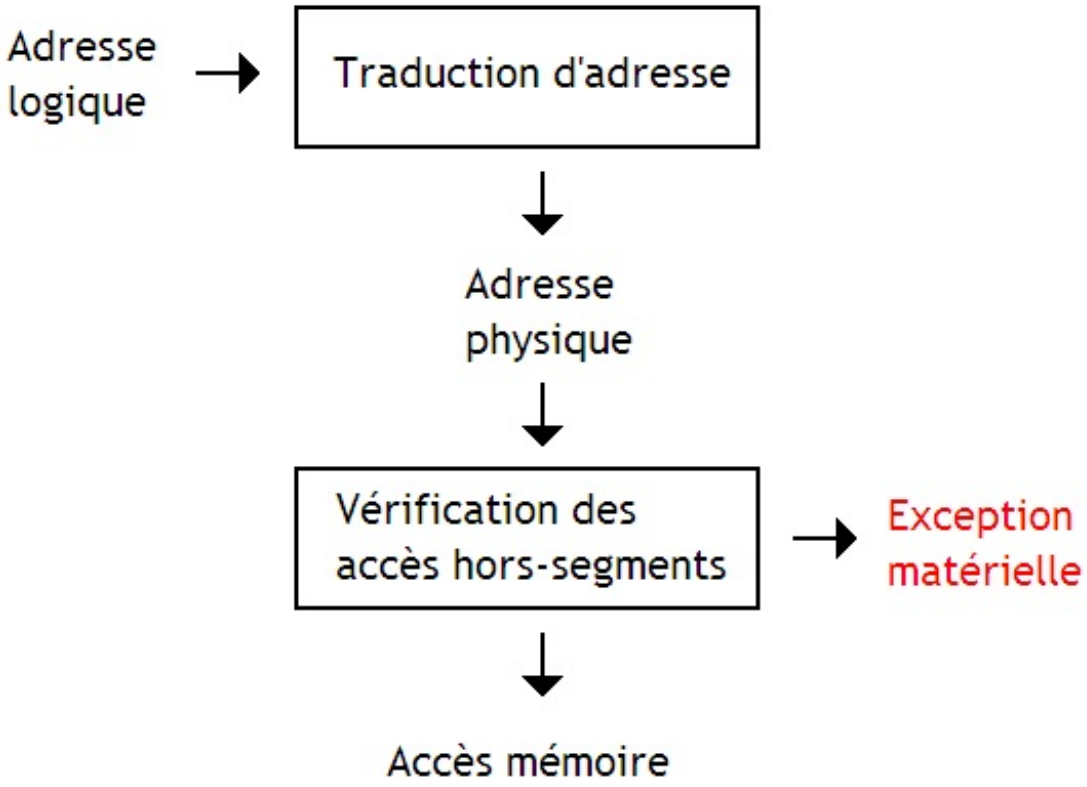
Cette gestion des droits sera prise en charge par la **MMU**, qui empêchera aux programmes d'aller écrire ou lire des données qu'ils n'ont pas le droit de toucher. Ces droits sont différents pour chaque programme et peuvent être différents pour la lecture et l'écriture : on peut ainsi autoriser à un programme de lire une partie de la mémoire, mais pas d'y écrire, ou autoriser lecture et écriture, ou interdire les deux.

Toute tentative d'accès à une partie de la mémoire non-autorisée déclenchera ce qu'on appelle une exception matérielle (rappelez-vous le chapitre sur les interruptions) qui devra être traitée par une routine du système d'exploitation. Généralement, le programme fautif est sauvagement arrêté et supprimé de la mémoire, et un message d'erreur est affiché à l'écran.

La segmentation va ainsi permettre d'implanter des mécanismes permettant de supprimer ces manipulations dangereuses. Bien sûr, on pourrait les implémenter sans avoir à utiliser la segmentation, mais ce serait plus difficile. On va voir comment la segmentation nous permettra de gérer facilement cette protection de la mémoire.

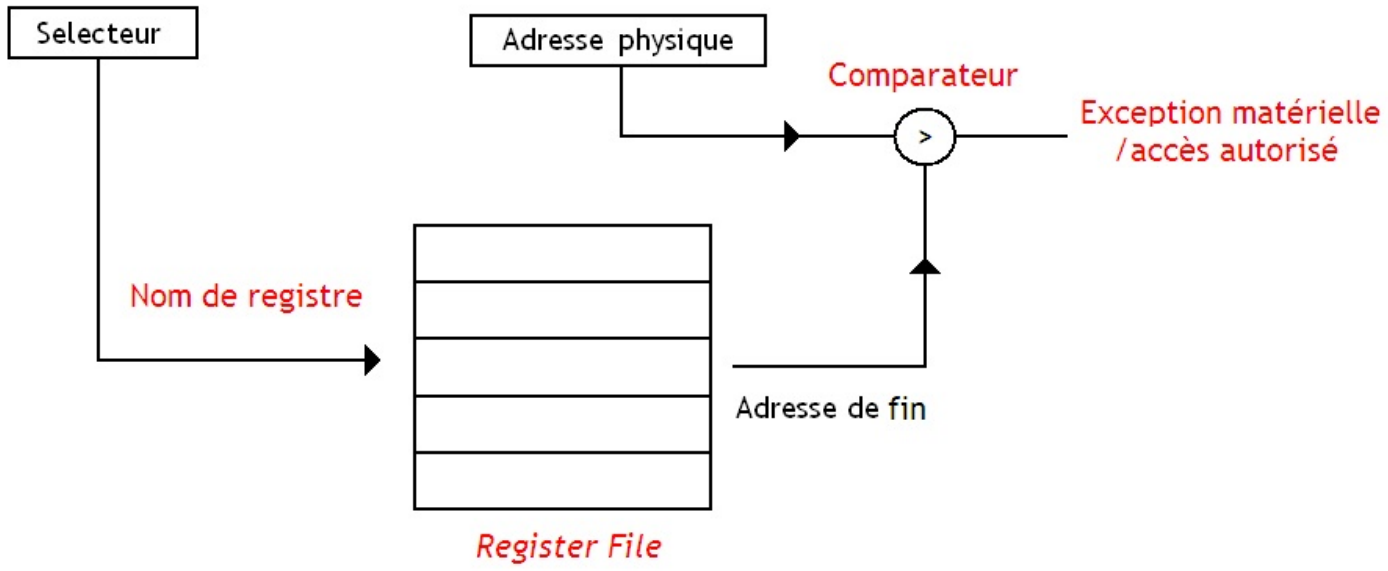
Gestion des accès hors segment

La première solution à ce problème est d'empêcher les accès qui débordent d'un segment. L'idée est de tester si l'adresse physique calculée par la MMU déborde au delà du segment. Si jamais l'accès déborde du segment, le processeur lève alors une exception matérielle, qui est traitée par le système d'exploitation de l'ordinateur.

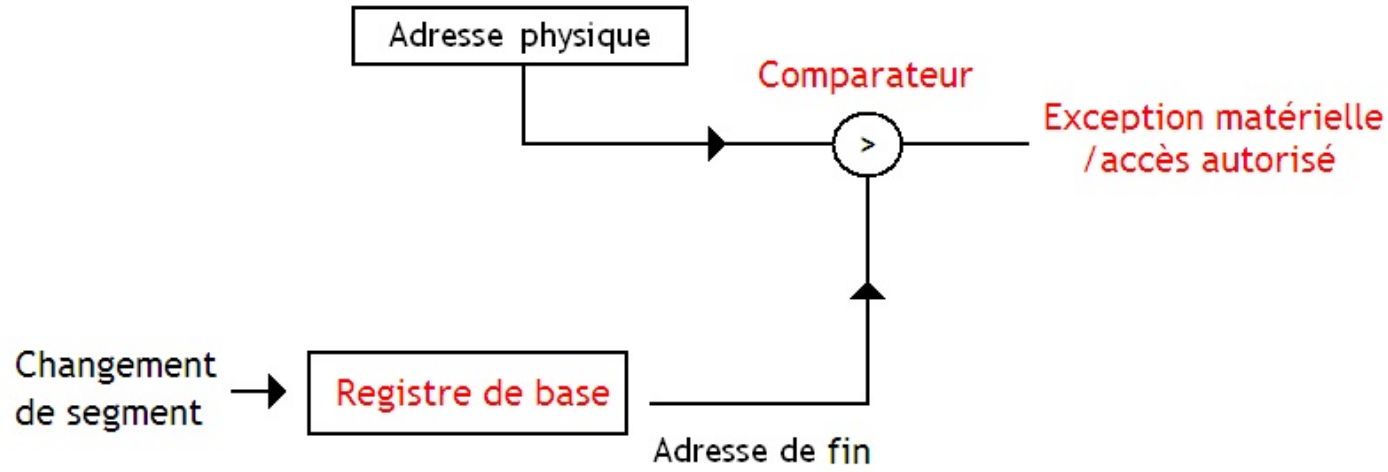


Pour effectuer cette vérification, rien de plus simple : il suffit de comparer l'adresse de fin de notre segment avec l'adresse à laquelle on veut accéder. Si cette adresse physique est plus grande que l'adresse de fin du segment, ça déborde. On peut aussi éviter d'avoir à comparer des adresses, et comparer l'*Offset* avec la longueur du segment. Si l'Offset est plus grand que la taille du segment, c'est qu'il y a un problème. Dans tous les cas, notre MMU devra incorporer un comparateur.

Pour cela, il y a une seule solution : notre MMU doit se souvenir non seulement de l'adresse du début de chaque segment, mais aussi des adresses de fin. Pour cela, on doit donc avoir une seconde table de correspondance qui associe le sélecteur d'un segment à son adresse de fin. Cette table de correspondance peut être placée en mémoire RAM. Mais encore une fois, il vaut mieux placer celle-ci dans les registres du processeur.



Plus haut, on a vu qu'implanter toute une table de correspondance complète dans le processeur était un gâchis de circuits. Au lieu de garder toute une table de correspondance dans des registres, on avait vu qu'on pouvait aussi se contenter d'un simple registre de base. Et bien il est aussi possible d'appliquer la même technique pour les adresses de fin. Au lieu d'avoir toute une table de correspondance complète, il vaut mieux réduire celle-ci à un simple **registre limite**. Celui-ci contient soit l'adresse de fin de notre segment, soit sa longueur.



Droits d'accès

Vient ensuite la gestion des droits d'accès en lecture et en écriture. Chaque segment se voit attribuer à sa création un certain nombre d'autorisations d'accès. Ces droits indiquent si l'on peut : lire ou écrire dans un segment, mais aussi considérer que celui-ci contient des données ou des instructions : on peut ainsi exécuter le contenu d'un segment ou au contraire interdire cette exécution.

Par exemple, le segment *Text* peut être exécutable : on peut considérer son contenu comme un programme, qui ne doit pas être modifié ou lu comme le serait une donnée : on peut le rendre *executable only*, et interdire de copier son contenu dans les registres généraux, ou d'écrire dedans. De même, on peut décider d'interdire de charger le contenu d'un segment dans le registre d'instruction ou le registre d'adresse d'instruction pour éviter d'exécuter des données (ce qui rend plus difficile certaines failles de sécurité ou l'exécution de certains virus).

De plus chaque segment est attribué à un programme en particulier : un programme n'a pas besoin d'accéder aux données localisées dans un segment appartenant à un autre programme, alors autant limiter les erreurs potentielles en spécifiant à quel programme appartient un segment.

Lorsqu'on veut exécuter une opération interdite sur un segment, il suffira à la MMU de déclencher une exception matérielle pour traiter l'erreur. Pour cela, pas de miracle : il faut retenir les autorisations pour chaque segment. Toutes ces informations sont rassemblées dans ce qu'on appelle un **descripteurs de segment**.

Celui-ci contient pour chaque segment des informations comme :

- son adresse de base ;
- son adresse de fin ou sa longueur ;
- les différentes autorisations de lecture ou d'écriture ;
- et d'autres choses encore.

Quand on décide d'accéder aux données ou aux instructions présentes dans un segment, ce descripteur sera chargé dans des registres du processeur (dont le fameux registre de base et le registre limite). Notre processeur pourra ainsi accéder à toutes les informations nécessaires pour gérer la protection mémoire et la traduction des adresse logiques en adresses physiques. Pour se simplifier la tache, les concepteurs de processeur et de système d'exploitation ont décide de regrouper ces descripteurs dans une portion de la mémoire, spécialement réservée pour l'occasion : la **table des descripteurs de segment**.

Comme pour les adresse de début et de fin de segment, les droits d'accès de nos programmes peuvent aussi être stocké dans la MMU, que ce soit dans un registre simple ou dans un *Register File* complet. Un circuit devra obligatoirement être intégré dans la MMU pour vérifier que l'instruction en cours est autorisée.

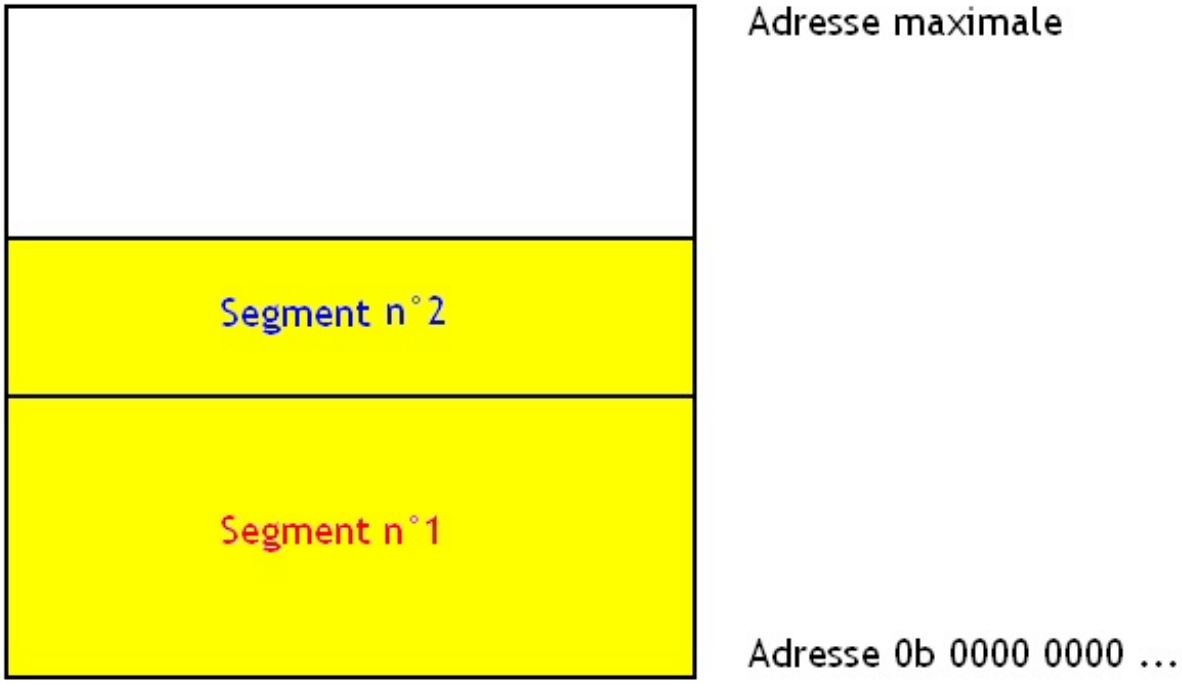
Allocation dynamique

Attention toutefois : nos segments peuvent avoir des tailles variables ! Certains segments peuvent ainsi grossir ou diminuer au fil du temps. Cela permet de réserver juste ce qu'il faut de mémoire au lancement d'un programme, et augmenter ou diminuer la quantité de mémoire réservée à celui-ci suivant les besoins.

Si notre programme a besoin de plus de mémoire quand il est en train de s'exécuter, le programme peut alors demander de grossir le segment qu'il est en train d'occuper, grâce à une interruption logicielle spécialement conçue pour. Bien sûr, quand un programme n'a plus besoin d'une portion de mémoire, il peut "dé-réserver" celle-ci et la rendre utilisable par notre système d'exploitation en diminuant la taille du segment qu'il occupe : on dit que notre programme libère la mémoire.

Quelques problèmes

Bien sûr, ça ne marche pas toujours. Imaginons le cas suivant : deux programmes sont lancés et sont stockés dans deux segments différents. Ces programmes vont alors régulièrement avoir besoin de mémoire et vont prendre de la mémoire quand ils en ont besoin. Imaginez qu'un programme aie tellement grossit qu'on en arrive à la situation suivante :



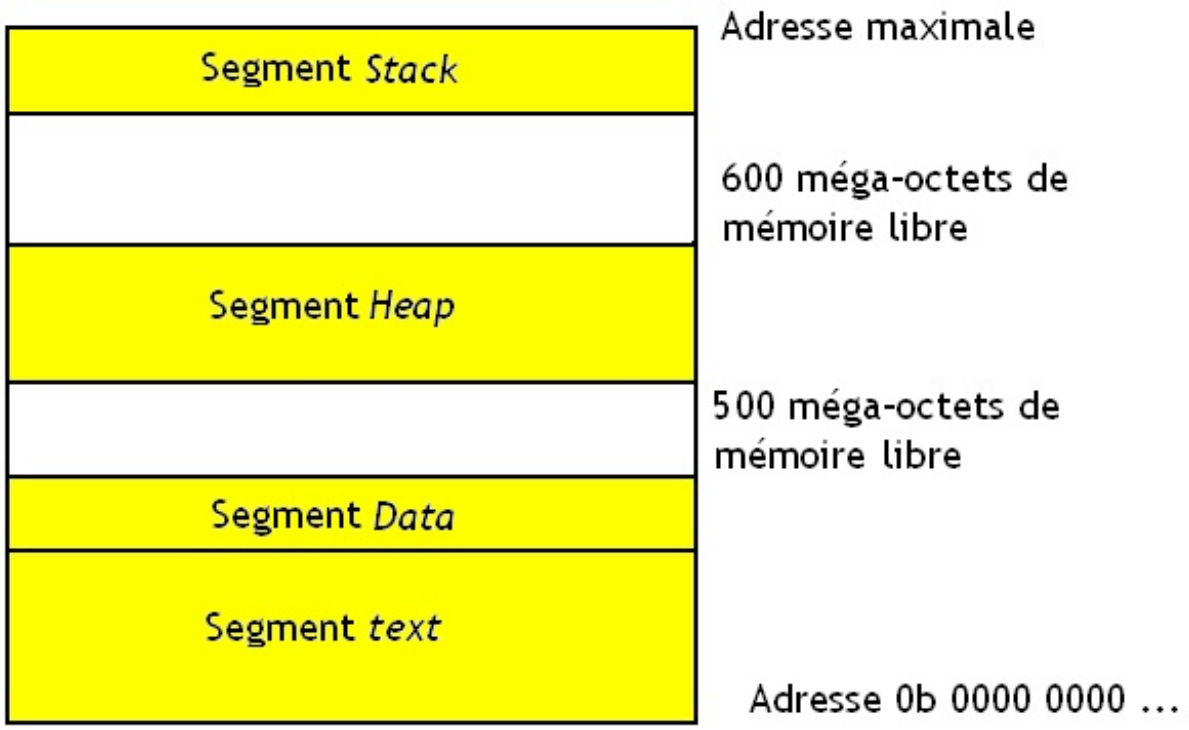
Imaginez maintenant que le programme N°1 aie besoin de plus de mémoire, que se passe-il ?

Je suppose que vous voyez bien qu'il y a un problème : il n'y a pas de mémoire libre à la suite du programme N°1, et son segment

ne peut pas grossir. Pour le résoudre, notre système d'exploitation va devoir déplacer au moins un programme dans la mémoire et réorganiser la façon dont ceux-ci sont répartis en mémoire afin de faire de la place à la suite du premier segment. Ce qui signifie que au moins un des deux segments sera déplacé : ça demande de faire beaucoup d'accès mémoire et prend donc pas mal de temps.

Fragmentation externe

La segmentation pose toutefois un petit problème : de la mémoire est gâchée un peu bêtement et est difficilement récupérable. Cela vient du fait que nos segments ne sont pas "collés les uns au autres", et qu'il existe des vides de mémoires, qu'on ne peut pas forcément remplir facilement. Pour expliquer la situation, on va prendre un exemple : on lance un programme, avec 4 segments, qui voient leur taille bouger au fil du temps et sont alloués par le système d'exploitation. Leur taille augmente, et on aboutit à la situation décrite dans le schéma ci-dessus.



Comme on le voit, il reste des vides de mémoires entre les segments. Ce n'est pas un mal : si des segments veulent augmenter leur taille, il leur reste un peu de marge.

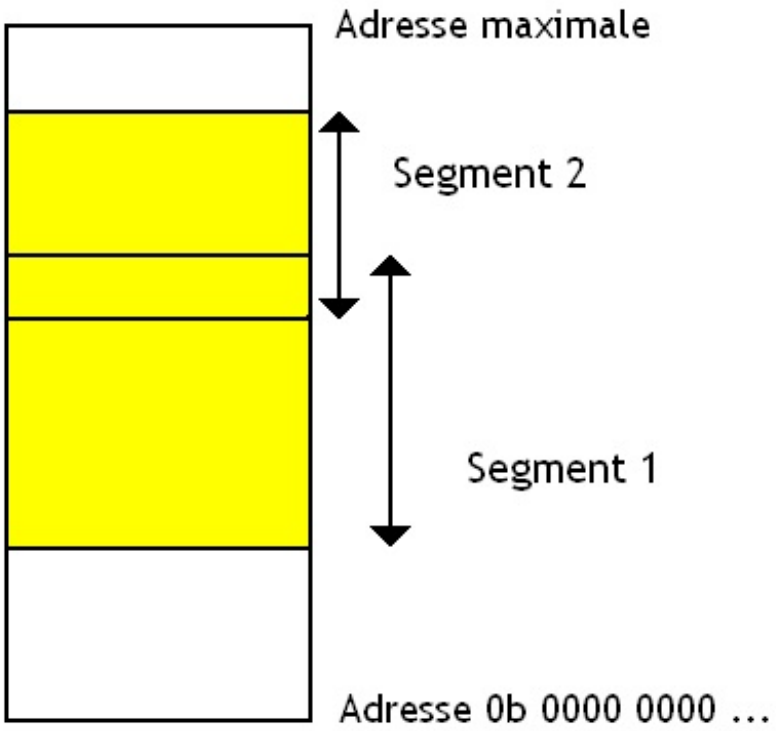
Et maintenant, dans la situation décrite par le schéma du dessus, imaginez qu'on lance un programme qui a besoin de 700 méga-octets de mémoire pour charger ses segments *text* et *data* a la suite. On aura suffisamment de mémoire libre pour les caser : on a bel et bien 1.1 gibi-octet de libre, mais aucun bloc de mémoire libre ne sera suffisamment gros pour répondre à la demande.

C'est ce qu'on appelle le phénomène de **fragmentation externe** : on dispose de suffisamment de mémoire libre, mais celle-ci est dispersée dans beaucoup de petits segments vides, qui peuvent difficilement stocker un segment à eux tout seuls.

Si aucun bloc de mémoire vide n'est suffisamment gros pour combler une demande d'un programme, notre système d'exploitation va devoir regrouper les morceaux de mémoire utilisés par les différents programmes et les déplacer pour créer des vides plus gros. En clair, il va devoir déplacer des segments entiers dans la mémoire, ce qui prend beaucoup de temps inutilement.

Partage de segments

Pour terminer cet aperçu de la segmentation, on peut signaler que des morceaux de segments peuvent être partagés. Rien ne l'empêche : il suffit de donner à deux segments des adresses de base et des longueurs convenablement étudiées pour.



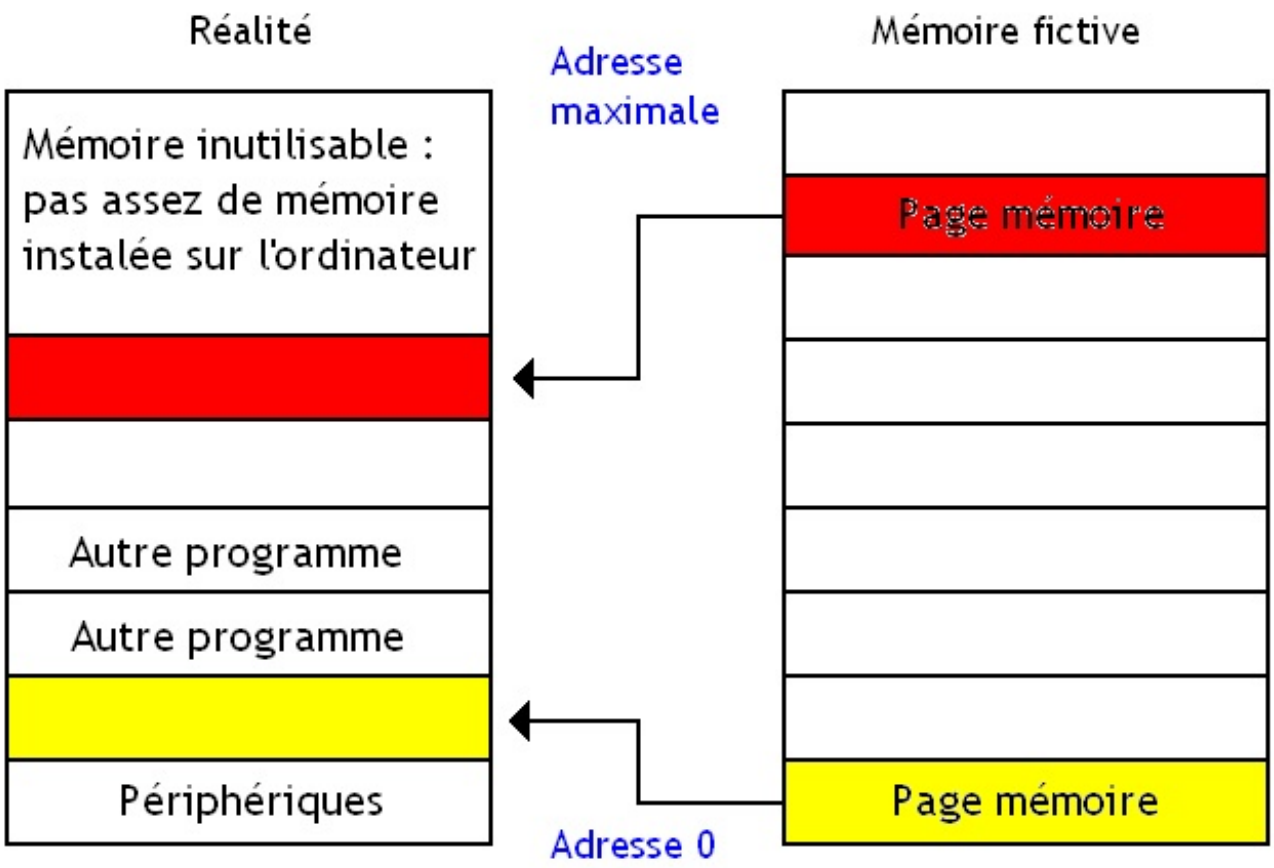
Pagination

De nos jours, la segmentation est considérée comme obsolète et n'est plus vraiment utilisée, malgré ses atouts de taille concernant la protection mémoire. On utilise à la place une autre technique de mémoire virtuelle nommée la **pagination**.

Que ce soit avec la segmentation ou avec la pagination, notre mémoire virtuelle et notre mémoire physique sont toujours découpées en gros blocs de données. La différence entre ces deux techniques tient dans la taille des blocs et ce qu'on met dedans. Avec la segmentation, nos segments avaient des tailles variables et étaient souvent utilisés pour stocker quelque chose de précis : un programme, des données, la pile, etc. Avec la pagination, tout ces blocs ont une taille fixe et ne sont pas organisés de façon vraiment logique. Ces blocs de mémoire de taille fixe sont appelés des **pages mémoires**.

Autre détail : la mémoire physique et la mémoire virtuelle sont découpées en pages. Ces pages sont toutes de la même taille, que ce soit en mémoire physique ou dans la mémoire fictive. Cette taille varie suivant le processeur, le système d'exploitation, et peut parfois être réglée manuellement. Cette taille tourne souvent autour de 4 kibi-octets : c'est la taille la plus couramment employée par les système d'exploitation comme Windows ou Linux, voire Mac OS.

Toute page mémoire de la mémoire fictive peut être placée à n'importe quelle page mémoire de la mémoire physique. Ainsi, une page dans la fausse mémoire sera placée dans une page de la mémoire physique, qui peut être n'importe laquelle. Une page en mémoire physique correspond donc à une page dans notre fausse mémoire.



Swapping

Vous aurez remarqués que notre mémoire physique contient moins de pages que la mémoire fictive. Pourtant, rien n'empêche d'utiliser plus de pages que notre mémoire contient : si on doit partager cette mémoire physique avec plusieurs programmes, ce genre de situation peut arriver, après tout. Il nous faut donc trouver un moyen de faire en sorte que cela ne pose pas de problème. La solution consiste à utiliser des mémoires de stockage comme mémoire d'appoint : si on besoin de plus de page mémoires que la mémoire physique n'en contient, alors certaines pages mémoires vont être déplacées sur le disque dur (ou toute autre mémoire de stockage) pour faire de la place.

Cela implique que certaines pages mémoires sont localisées sur le disque dur. Et l'on ne peut y accéder directement : on doit d'abord les charger dans la mémoire RAM, avant de pouvoir les modifier. Il faut donc les rapatrier en mémoire RAM, ce qui peut prendre du temps. Lorsque l'on veut traduire l'adresse logique d'une page mémoire qui a étè déplacée sur le disque dur, notre MMU ne va pas pouvoir associer l'adresse logique à une adresse en mémoire RAM. Elle va alors lever une **exception matérielle**, une sorte d'interruption que le processeur exécute automatiquement lorsque certains évènement arrivent. Cette exception matérielle déclenchera l'exécution d'une routine (souvent fournie par le systéme d'exploitation) qui rapatriera notre page en mémoire RAM.

Remplacement des pages mémoires

Charger une donnée depuis le disque dur ne pose aucun problème tant qu'il existe de la mémoire RAM disponible. On charge alors la donnée dans une page qui est inoccupée et vierge de données : c'est juste lent, sans plus. Mais il se peut que la RAM disponible soie insuffisante pour accueillir la page mémoire à charger depuis le disque dur (quand toute le RAM est pleine, par exemple). Dans ce cas, il faut déplacer le contenu d'une page mémoire localisée sur le disque dur pour faire de la place pour l'autre page à charger depuis le disque dur.

Tout cela est effectué par la fameuse routine du système d'exploitation dont j'ai parlé plus haut. Il existe différents algorithmes qui permettent de décider quelle donnée supprimer de la RAM. Ces algorithmes ont une importance capitale en terme de performance. Si on supprime une donnée dont on aura besoin dans le futur, il faudra recharger celle-ci, et donc exécuter une interruption, accéder au disque dur, charger la page, etc. Et cela prend du temps induisant une perte de performance. Pour éviter cela, le choix de la page doit être fait avec le plus grand soin. Il existe divers algorithmes pour cela.

Algorithmes

Ces algorithmes sont les suivants :

- Aléatoire : on choisit la page au hasard.
- FIFO : on supprime la donnée qui a été chargée dans la mémoire avant toute les autres.
- LRU : on supprime la donnée qui été lue ou écrite pour la dernière fois avant toute les autres.
- LFU : on vire la page qui est lue ou écrite le moins souvent comparé aux autres.
- Et il en existe d'autres encore, mais on ne va pas en parler ici.

Si vous voulez vous renseigner un peu plus sur le sujet, allez voir sur cette page Wikipédia : Page Replacement Algorithms.

Locaux versus Globaux

Ces algorithmes ont chacun deux variantes : une locale, et une globale. Avec la version locale, la page qui va être rapatriée sur le disque dur est une page réservée au programme qui est la cause du Page Miss. Avec la version globale, le système d'exploitation va choisir la page à virer parmi toutes les pages présentes en mémoire vive.

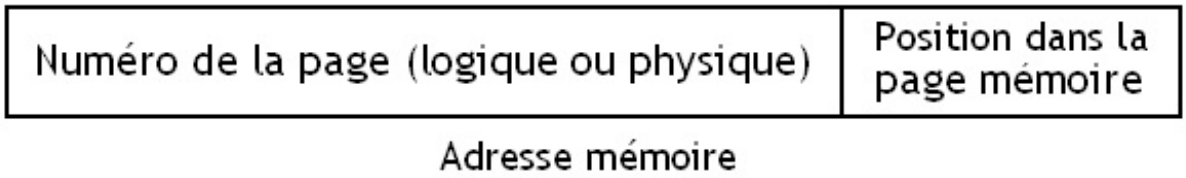
Pinning

Petite remarque : sur la majorité des systèmes d'exploitation, il est possible d'interdire le rapatriement de certaines pages mémoires sur le disque dur. Ces pages restent alors en mémoire RAM durant un temps plus ou moins long. Certaines restent même en RAM de façon permanente. Cette possibilité est très utile pour les programmeurs qui conçoivent des systèmes d'exploitation. Par exemple, cela permet de gérer les vecteurs d'interruptions assez simplement. Pour donner un exemple, essayez d'exécuter une interruption de gestion de Page Miss alors que la page contenant le code de l'interruption est placée sur le disque dur.

Translation d'adresse

Par contre, le contenu d'une page en mémoire fictive est rigoureusement le même que le contenu de la page correspondante en mémoire physique. Deux données qui se suivent à l'intérieur d'une page de la mémoire fictive se suivront dans une page de la mémoire réelle. Ainsi, on peut parfaitement localiser une donnée dans une page par un numéro, qui sera le même que cette page soit la page en mémoire physique ou la page de la fausse mémoire. Pour numéroter une case mémoire à l'intérieur d'une page, on utilise quelques bits de poids fort de l'adresse (physique ou logique).

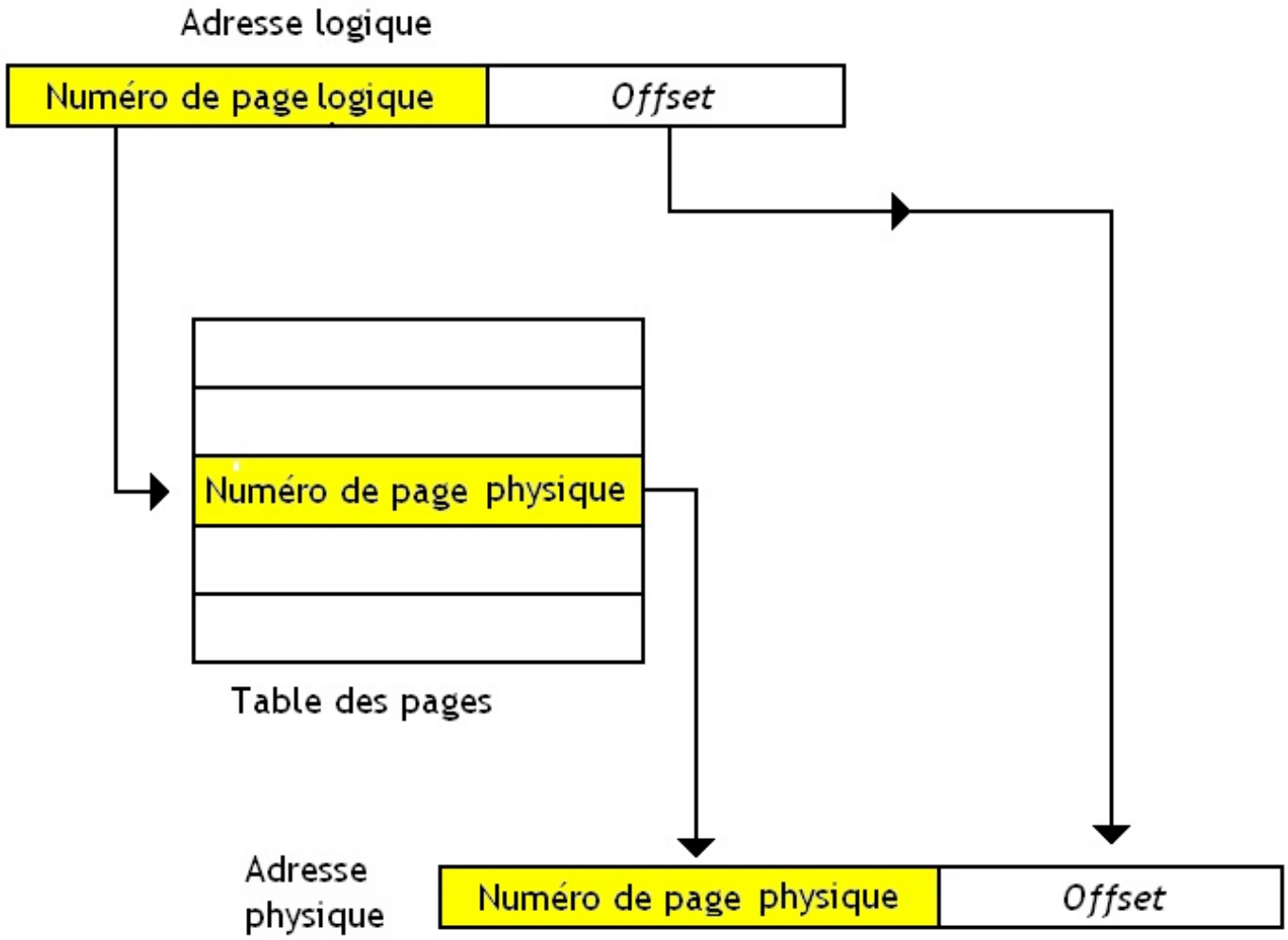
Une adresse (qu'elle soit logique ou physique) se décompose donc en deux parties : une partie qui identifie la page dans la mémoire (fictive ou physique suivant la nature de l'adresse), et un numéro permettant d'identifier la donnée dans la page.



Traduire notre adresse logique en adresse physique consiste simplement à remplacer le numéro de la page logique en un numéro de page physique.

Page Table Entry

Pour faire cette traduction, on n'a pas vraiment le choix : il faut se souvenir des correspondances entre page en mémoire physique et page en mémoire fictive pour toutes les pages. Ces correspondances sont stockées dans une sorte de table, nommée la **table des pages**. Ainsi, pour chaque numéro (ou chaque adresse) de page logique, on stocke le numéro ou l'adresse de base de la page correspondante en mémoire physique.



Il faut préciser que cette table des pages est unique pour chaque programme : chaque programme manipule les mêmes adresses virtuelles. Ce qui fait que deux programmes peuvent manipuler des adresses logiques identiques dans leur mémoire fictive qui leur est attribuée. Pourtant, les données correspondant à ces adresses logiques seront différentes et seront stockées dans des adresses mémoires physiques différentes. Pour éviter les problèmes, on n'a pas le choix : il faut soit utiliser des tables des pages différentes pour chaque programme.

Cette table des pages est souvent stockée dans la mémoire RAM, à un endroit bien précis, connu du processeur. Accéder à la mémoire nécessite donc d'accéder d'abord à la table des pages en mémoire, puis de calculer l'adresse de notre donnée, et enfin d'accéder à la donnée voulue.

TLB

Pour éviter d'avoir à lire la table des pages en mémoire RAM à chaque accès mémoire, les concepteurs de processeurs ont décidés d'implanter une petite mémoire cache dans le processeur, qui stocke une partie de la table des pages : généralement, on conserve dans ce cache le morceau de la table des pages qui sert à traduire les dernières adresses ayant étè accédées. Cette mémoire cache s'appelle le ***Translation Lookaside Buffer***, aussi appelé ***TLB***.

Ainsi, notre processeur va vérifier si le morceau de la table des pages stocké dans ce TLB permet de calculer l'adresse physique à laquelle accéder. Si c'est le cas, le processeur n'a pas à accéder à la mémoire RAM et va lire directement la donnée depuis ce TLB. Mais dans le cas contraire, l'accès à la RAM est inévitable.
Cet accès est géré de deux façon :

- soit le processeur gère tout seul la situation ;
- soit il délègue le traitement de la situation à un programme particulier ou au système d'exploitation.

Dans le premier cas, le processeur est conçu pour lire lui même le contenu de la page des tables en mémoire et y trouver la bonne correspondance dans celle-ci. Une fois trouvée, le processeur va alors copier celle-ci dans le TLB et recalculer l'adresse physique. Dans le second cas, si l'adresse cherchée n'est pas dans le TLB, le processeur va lever une exception matérielle qui exécutera une routine d'interruption chargée de gérer la situation.

Allocation dynamique

Comme avec la segmentation, nos programmes peuvent demander au système d'exploitation l'accès à un peu plus de mémoire de temps à autre. Si jamais un programme a besoin de plus de mémoire, notre système d'exploitation va lui réserver une page mémoire supplémentaire, qu'il pourra manipuler à loisir. Bien sûr, si notre programme n'a plus besoin de la page qui lui a été confiée, il peut la libérer : elle sera alors disponible pour n'importe quel autre programme qui en aurait besoin.

Il faut remarquer que le problème mentionné avec des segments consécutifs, "qui se touchent", n'apparait pas avec la pagination. On peut fragmenter notre programme dans des pages dispersées dans la mémoire sans aucun problème, ce qui fait qu'on n'est pas obligé de déplacer des pages mémoires pour compacter la mémoire libre. Cela fait qu'avec la pagination, il n'y a pas vraiment de fragmentation externe, vu qu'on peut fragmenter notre programme dans des pages différentes pour utiliser au mieux la mémoire libre disponible au lieu de tout devoir caser dans un seul gros segment.

Mais cela n'empêche pas certaines pertes de mémoire utilisable. En effet, quand un programme demande un surplus de mémoire, le programme qui gère la gestion de la mémoire (c'est souvent le système d'exploitation), va réserver une page mémoire complète : il ne peut pas faire autrement, même si notre programme n'a besoin que de quelques octets. Si un programme a besoin d'une quantité plus petite que ce que la page lui offre, une partie de notre page sera réellement utilisée pour stocker des données. Par exemple, un programme voulant réserver 2 kibi-octets se verra attribuer le double (en supposant une page de 4 kibi-octets) : de l'espace mémoire est perdu.

Et la même chose arrive lorsque notre programme a besoin d'une quantité de mémoire qui n'est pas multiple de la taille d'une page. Si un programme a besoin de 5 kibi-octets, il se verra attribuer deux pages de 4 kibi-octets : 3 kibi-octets ne serviront à rien. Ce genre de pertes du au fait que la mémoire est allouée par pages s'appelle la **fragmentation interne**.

Protection mémoire

La pagination permet elle aussi de protéger la mémoire et son contenu de manipulations potentiellement dangereuses sur nos pages ou sur leur contenu. Avec la pagination, on peut parfaitement autoriser ou interdire la lecture ou l'écriture, voir l'exécution du contenu d'une page. Pour cela, il suffit de rajouter des bits dans la table des pages : chaque numéro de page virtuelle/physique se verra attribuer des droits, représentés par des bits d'une certaine valeur. Suivant la valeur de ces bits, la page sera accessible ou non, en lecture, en écriture, exécutable, etc.

De plus chaque page est attribuée à un programme en particulier : un programme n'a pas besoin d'accéder aux données localisées dans une page réservée par un autre programme. Autant limiter les erreurs potentielles en spécifiant à quel programme appartient une page. Pour cela, des bits permettant d'identifier le programme possesseur de la page sont ajouté en plus des adresses et des bits de gestion des droits d'accès en lecture/écriture dans la table des pages.

Les mémoires caches

Les mémoires caches sont des mémoires intercalées entre le mémoire et un processeur ou un périphérique. Ces mémoires peuvent contenir assez peu de données et sont assez rapides. Elles sont souvent fabriquées avec de la mémoire SRAM.

Ces mémoires sont facultatives : certains processeurs ou périphériques se passent complétement de mémoires caches alors que d'autres en ont plusieurs. Dans ce chapitre, on va parler des caches dédiés au processeur, et pas des caches présents dans les périphériques.

Mais à quoi sert ce cache dédié au processeur et pourquoi on en a besoin ?

Sans lui, on se croirait à l'age de pierre tellement nos PC seraient lents ! Il faut dire que la mémoire d'un ordinateur est quelque chose de vraiment très lent, comparé à un processeur. Cela pose problème lorsque le CPU cherche à accéder à la mémoire (aussi bien en lecture qu'en écriture). Le temps mit pour enregistrer ou récupérer des données depuis la mémoire est du temps durant lequel le processeur n'exécute pas d'instructions (sauf cas particuliers impliquant un pipeline et l'exécution *out-of-order*). Il a bien fallu trouver une solution pour diminuer le plus possible ce temps d'attente, et on a décidé d'intercaler une mémoire entre le CPU et la mémoire. Comme cela, le processeur accède directement à une mémoire cache très rapide plutôt que d'accéder à une mémoire RAM qui répondra de toute façon trop tard.

Accés au cache

Notre mémoire cache est, comme toutes les autres mémoires, divisées en cases mémoires, qu'on peut modifier individuellement. Dans un cache, ces cases mémoires sont regroupées en blocs de taille fixe qu'on appelle des **lignes de cache**. Généralement, ces blocs ont une taille assez grosse comparé aux cases mémoires : cela peut varier de 64 à 256 octets.

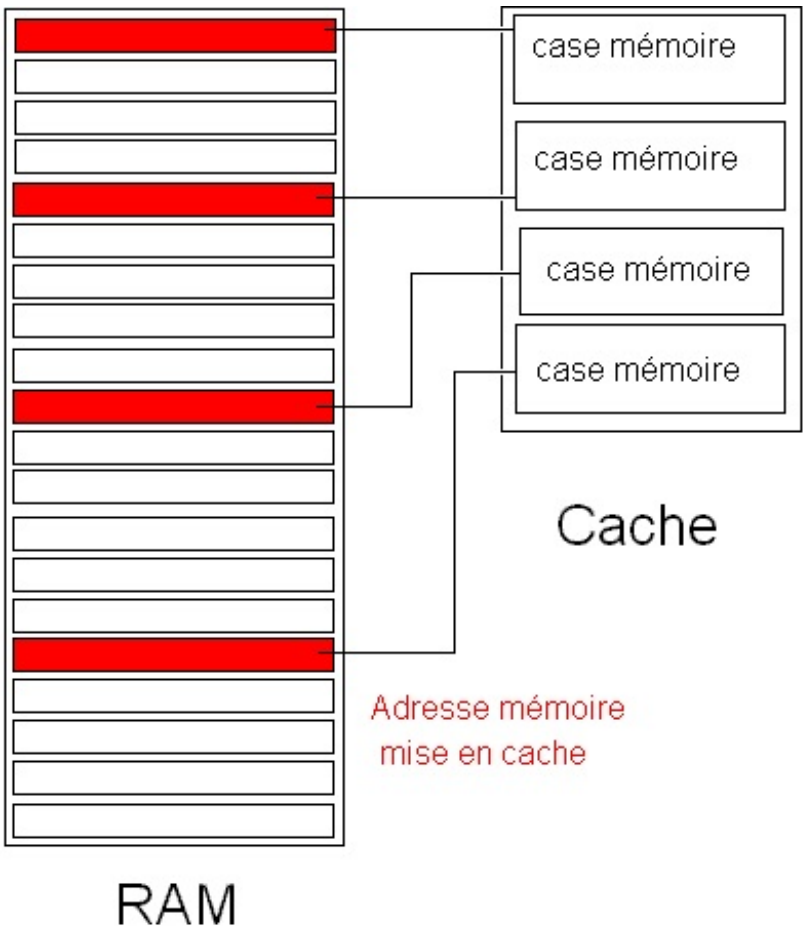
Sur la majorité des caches actuels, on est obligé de transférer des donnée entre le cache et la mémoire, ligne de cache par ligne de cache. Il est impossible de charger seulement un morceau d'une ligne de cache depuis la mémoire. Les transferts se font lignes de caches par lignes de cache. Par contre, chaque case mémoire d'une ligne de cache peut être accédée individuellement par le processeur, que ce soit en lecture ou en écriture. En clair, on peut modifier une case mémoire à l'intérieur d'une ligne sans problème. Le regroupement en lignes de cache ne compte que pour les transferts entre mémoire RAM et cache, pas pour les transferts entre cache et registres.

Accès au cache

Comme je l'ai dit et répété plus haut, une case mémoire du cache ne possède pas d'adresse. Aussi bizarre que cela puisse paraitre, nos mémoires caches ne sont pas les seules mémoires à ne pas pouvoir être adressables, mais passons. Cela a une conséquence : notre processeur ne va pas accéder directement à la mémoire cache. Il peut juste demander une lecture ou écriture dans la RAM, qui aboutira ou non à une lecture/écriture à partir du cache.

Notre processeur va donc accéder à la mémoire RAM et cet accès mémoire sera intercepté par le cache, qui vérifiera si la donnée demandée est présente ou non dans le cache. Tout se passe comme si le cache n'existait pas pour le processeur. Et vu que notre processeur ne peut pas gérer lui-même le cache, on se doute bien que notre cache se "gère tout seul" : il possède un ou plusieurs circuits électroniques conçus pour gérer le cache automatiquement.

Les données présentes dans le cache sont été (pré)chargées depuis la mémoire : toute donnée présente dans le cache est la copie d'une donnée en mémoire RAM.



On a donc une correspondance entre une ligne de cache et une adresse mémoire, mémorisée par les circuits électroniques qui s'occupent de la gestion de la mémoire cache.

Contenu de la ligne de cache	Adresse mémoire correspondante
0100 1110 1001 1011	1001 0000 0000 0001
1111 0000 1111 1111	0110 1111 0101 1011
1100 1100 1010 1001	0000 0000 1000 0000
...	...

Lorsqu'un processeur cherche à accéder à la mémoire (que ce soit pour une lecture ou une écriture), celui-ci va envoyer l'adresse mémoire à laquelle il veut accéder vers un circuit qui permet d'écrire ou de lire en mémoire RAM. Cette adresse va d'abord être interceptée par les circuits de gestion du cache, qui vont vérifier si cette adresse mémoire correspond à une ligne de cache.

Si c'est le cas, la donnée voulue est présente dans le cache : on a un ***cache hit*** et on accède à la donnée depuis le cache.

Sinon, le donnée n'est pas dans le cache : on a un ***cache miss*** et on est obligé d'accéder à la RAM ou de recopier notre donnée de la RAM dans le cache.

Le nombre de *cache miss* par nombre d'accès mémoire est appelé le ***hit ratio***. Plus celui-ci est élevé, plus on accède au cache à la place de la RAM et plus le cache est efficace. Ce *hit ratio* varie beaucoup suivant le processeur, l'organisation des caches, ou le programme en cours d'exécution : de nombreux paramètres peuvent influencer celui-ci.

Écriture dans un cache

La lecture d'une donnée présente dans le cache ne pose aucun problème (sauf cas particuliers) : il suffit d'accéder au contenu du

cache, et on reçoit la donnée voulue assez rapidement. Mais les choses changent pour l'écriture. L'écriture dans un cache fait face à diverses situations, qu'il faut gérer au mieux. Pour gérer certaines situations embarrassantes, deux stratégies d'écritures sont couramment implémentées dans les circuits de gestion du cache.
Ces deux stratégies sont

- le *Write Back* ;
- et le *Write Throught*.

Write back

Si la donnée qu'on veut mettre à jour est présente dans le cache, on écrit dans celui-ci sans écrire dans la mémoire RAM : on attend que la donnée soit effacée du cache pour l'enregistrer en mémoire. Cela évite de nombreuses écritures mémoires inutiles.

Mais peut poser des problèmes dans les architectures multiprocesseurs ! Si deux processeurs doivent manipuler la même donnée, des cas litigieux peuvent survenir. Imaginons que deux processeurs manipulent une donnée : ceux-ci ont une copie de la donnée dans leur cache qu'ils manipulent et modifient à loisir. Si un processeur modifie cette copie de la donnée et que celle-ci est enregistrée dans son cache ou en mémoire, elle sera alors différente de celle présente dans le cache de l'autre processeur. Ce qui fait qu'un processeur peut continuer à manipuler une donnée périmée qui vient d'être mise à jour par l'autre processeur. Hors, un processeur doit toujours éviter de se retrouver avec une donnée périmée et doit toujours avoir la valeur correcte dans ses caches : cela s'appelle la **cohérence des caches**.

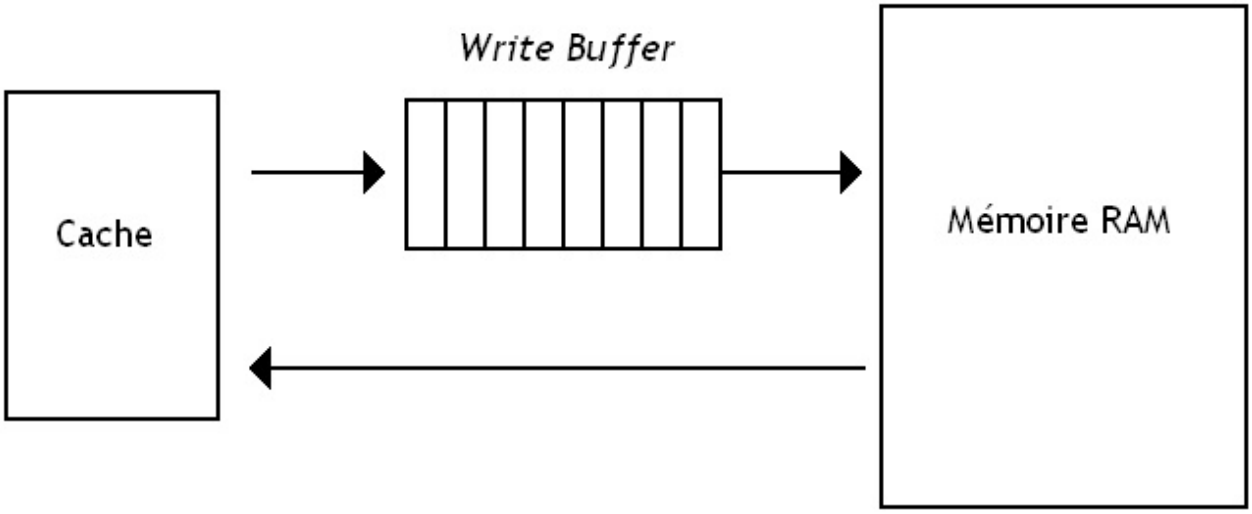
La seule solution pour arriver à un résultat correct est d'utiliser des mécanismes permettant de faire en sorte que ce genre de cas n'arrivent pas. Ces mécanismes se chargent de déterminer quelles sont les données périmées, et ensuite les remplacer par des données valides. Ces mécanismes se chargent de faire mettre à jour les données dans les différents caches, en copiant les données mises à jour du cache d'un processeur vers un autre, en passant éventuellement en se servant de la mémoire comme intermédiaire. Les caches *Write Back* rendent plus difficile l'implantation de ces mécanismes de gestion de la cohérence des caches. C'est pourquoi il existe un autre type de cache : le cache *Write Trought* ; mieux adapté à ce genre de situations.

Write Through

Toute donnée écrite dans le cache est écrite en même temps dans la mémoire RAM. Ces caches sont appelés ***No Write Allocate*** : si on souhaite écrire en mémoire une donnée qui n'a pas été préchargée dans le cache, on écrit en mémoire RAM, mais pas dans le cache. Leur utilité ? Les ordinateurs avec plusieurs processeurs, comme dit plus haut. Mais cela a fatalement un cout en terme de performance.

Ces caches ont tendance commettre beaucoup d'écritures dans la mémoire RAM, ce qui peut saturer le bus reliant le processeur à la mémoire. De plus, on ne peut écrire dans ces caches lorsqu'une écriture en RAM a lieu en même temps : cela forcerait à effectuer deux écritures simultanées, en comptant celle imposée par l'écriture dans le cache. Dans ces conditions, on doit attendre que la mémoire RAM soit libre pour pouvoir écrire dans notre cache.

Pour éviter ces temps d'attentes, certains processeurs intègrent un ***Write Buffer*** : une espèce de mémoire tampon dans laquelle on place temporairement les données à transférer du cache vers la RAM en attendant que la RAM soit libre. Ces données à écrire sont placées dans l'ordre d'arrivé dans ce *Write Buffer* et sont automatiquement écrites en mémoire RAM quand celle-ci est libre. On n'a pas à se soucier du fait que la mémoire soit occupée vu qu'on écrit les données à écrire non pas en RAM, mais dans ce *Write Buffer* : on peut continuer à écrire dedans tant que celui-ci n'est pas plein, évitant les temps d'attente dus à la RAM.



Par souci d'efficacité, les caches *Write Trough* implémentent parfois des techniques de *Write Combining* dans le *Write Buffer*. Le *Write Combining* est une technique assez simple à comprendre : si jamais deux écritures sont en attente dans le *Write Buffer*, seule la plus récente est prise en compte, et l'ancienne est abandonnée.

Comme ça, au lieu d'écrire une donnée avant de la remplacer immédiatement après par une autre, autant écrire directement la donnée la plus à jour et ne pas écrire l'ancienne inutilement. Cela fait un peu de place dans le *Write Buffer*, et lui permet d'accumuler plus d'écriture avant de devoir bloquer le cache. Et oui, une fois le *Write Buffer* plein, le cache n'est plus accessible en écriture : il n'y a plus de place pour une écriture supplémentaire. On doit donc bloquer le cache, histoire d'attendre qu'il y aie un peu de place dans le *Write Buffer* pour accueillir une nouvelle écriture.

Cache bloquant et non-bloquant

Un **cache bloquant** est un cache auquel le processeur ne peut pas accéder après un cache miss. Il faut alors attendre que la donnée voulue soit lue ou écrire en la RAM avant de pouvoir utiliser de nouveau le cache. Un **cache non-bloquant** n'a pas ce problème. On peut l'utiliser même immédiatement après un cache miss. Cela permet d'accéder à la mémoire cache en attendant des données en provenance de la mémoire. Tous les caches non-bloquants peuvent ainsi permettre de démarrer une nouvelle lecture ou écriture alors qu'une autre est en cours. On peut ainsi exécuter plusieurs lectures ou écritures en même temps : c'est ce qu'on appelle du ***Memory Level Parallelism***. Ces caches non-bloquants peuvent aussi permettre de pipeliner nos accès au cache.

Mais cela ne marche que tant qu'on ne veut pas trop faire d'accès au cache : au-delà d'un certain nombre, le cache va saturer et va dire "stop". Il ne peut supporter qu'un nombre limité d'accès mémoires simultanés (pipelinés).

Pour votre information, le *Write Buffer* permet à un cache *Write Trough* d'être non-bloquant ! En effet, sans lui, le cache serait immobilisé à chaque écriture : on devrait copier la donnée qui a étè écrite dedans du cache vers la mémoire, ce qui nécessite d'utiliser le cache. Avec le *Write Buffer*, la donnée à écrire en RAM est lue non pas depuis le cache,mais depuis le *Write Buffer*, ce qui libère le cache et le rend accessible aussi bien en lecture qu'en écriture. C'est pour cela qu'on les a aussi ajouté sur les caches Write Back.

Localité spatiale et temporelle

Mais pourquoi les caches fonctionnent ?

La question peut paraitre complètement stupide, mais pour expliquer cela correctement, il faut prendre conscience de quelques propriétés que beaucoup de programmes ont tendance à respecter.

Localité temporelle

Un programme a tendance à réutiliser les instructions et données qui ont étés accédées dans le passé : c'est la **localité temporelle**. Bien évidement, cela dépend du programme, de la façon dont celui-ci est programmé et accède à ses données et du traitement qu'il fait, mais c'est souvent vrai en général.

Lorsqu'on exécute à une instruction ou qu'on accède à donnée pour le première fois, celle-ci (l'instruction ou la donnée) n'a pas

encore été chargée dans le cache. Le *cache miss* est inévitable : ce genre de *cache miss* s'appelle un *Cold Miss*. Mais on peut être presque sûr que cette donnée sera réutilisée plus tard, et on pourra alors utiliser la version de la donnée chargée dans le cache au lieu d'accéder en mémoire. Il suffit alors de garder cette donnée dans le cache : d'après le principe de localité temporelle, elle sera surement réutilisée plus tard et sera alors disponibles dans le cache.

Évidement, cette technique à des limites. Si on doit accéder à beaucoup de données, le cache fini par être trop petit pour conserver les anciennes données : de nouvelles données sont ajoutés dans ce cache au fil du temps, et le cache doit bien finir par faire de la place en supprimant les anciennes données (qui ont peu de chances d'être réutilisées). Ces anciennes données présentes en cache, qui peuvent être accédées plus tard, devront céder la place à d'autres. Elles vont quitter le cache et tout prochain accès à cette donnée mènera à un *cache miss*. C'est ce qu'on appelle un *Capacity Cache Miss*.

Les seules solutions pour éviter cela consistent à augmenter la taille du cache, faire en sorte que notre programme prenne moins de mémoire cache et améliorer la localité du programme exécuté. Sachez qu'un programmeur peut parfaitement tenir compte de la localité spatiale lorsqu'il programme pour gagner énormément en performance. Pour donner un exemple, les boucles sont des structures de contrôle qui respectent le principe de localité temporelle : les instructions d'une boucle seront ré-exécutées plusieurs fois.

Localité spatiale

Autre propriété : un programme qui s'exécute sur un processeur à tendance à utiliser des instructions et des données qui ont des adresses mémoires très proches, c'est la **localité spatiale**.

Pour donner un exemple, les instructions d'un programme sont placées en mémoire dans l'ordre dans lequel on les exécutés : la prochaine instruction à exécuter est souvent placée juste après l'instruction en cours (sauf avec les branchements). La localité spatiale est donc respectée tant qu'on a pas de branchements qui renvoient assez loin dans la mémoire (appels de sous-programmes). De plus, on ne charge pas des données individuelles dans notre cache, mais des lignes de cache complètes. On charge donc la donnée voulue, mais aussi des données/instructions située dans des adresses mémoires proches : cela permet de gros gains lorsque le principe de localité spatiale est respecté.

L'influence du programmeur

De nos jours, le temps que passe le processeur à attendre la mémoire devient de plus en plus un problème au fil du temps, et gérer correctement le cache est une nécessité, particulièrement sur les processeurs multi-cores. Il faut dire que la différence de vitesse entre processeur et mémoire est tellement importante que les Cache Miss sont très lents comparées aux instructions machines usuellement employées : alors qu'une simple addition ou multiplication va prendre entre 1 et 5 cycles d'horloge, un cache miss fera plus dans les 400-1000 cycles d'horloge. Tout ce temps sera du temps de perdu que notre processeur tentera de remplir avec des instructions ayant leurs données disponibles (dans un registre voire dans le cache si celui-ci est non-bloquant), mais cela a une efficacité limitée. Autant dire que supprimer des *Caches Miss* sera beaucoup plus efficace que virer des instructions de calcul normales.

Bien évidement, optimiser au maximum la conception des caches et de ses circuits dédiés améliorera légèrement la situation, mais n'en attendez pas des miracles. Il faut dire qu'il n'y a pas vraiment de solutions anti-*Cache Miss* qui soit facile à implémenter. Par exemple, changer la taille du cache pour contenir plus de données aura un effet désastreux sur son temps d'accès qui peut se traduire par une baisse de performance. Par exemple, les processeurs *Nehalem* d'Intel ont vus leurs performances dans les jeux vidéos baisser de 2 à 3 % malgré de nombreuses améliorations architecturales très évoluées : la latence du cache L1 avait augmentée de 2 cycles d'horloge, réduisant à néant de nombreux efforts d'optimisations architecturales.

Non, une bonne utilisation du cache (ainsi que de la mémoire virtuelle) repose en réalité sur le programmeur qui doit prendre en compte les principes de localités vus plus haut dès la conception de ses programmes. La façon dont est conçue un programme joue énormément sur sa localité spatiale et temporelle. Un programmeur peut parfaitement tenir compte du cache lorsqu'il programme, et ce aussi bien au niveau :

- de son algorithme : on peut citer l'existence des algorithmes *cache oblivious* ;
- du choix de ses structures de données : un tableau est une structure de donnée respectant le principe de localité spatiale, tandis qu'une liste chainée ou un arbre n'en sont pas (bien qu'on puisse les implémenter de façon à limiter la casse);
- ou de son code source : par exemple, le sens de parcourt d'un tableau multidimensionnel peut faire une grosse différence.

Cela permet des gains très intéressants pouvant se mesurer avec des nombres à deux ou trois chiffres. Je vous recommande, si vous êtes programmeur, de vous renseigner le plus possible sur les optimisations de code ou algorithmiques qui concernent le cache : il vous suffira de chercher sur Google. Quoiqu'il en soit, il est quasiment impossible de prétendre concevoir des programmes optimisés sans tenir compte de la hiérarchie mémoire. Et cette contrainte va se faire de plus en plus forte quand on devra passer aux architectures multicœurs. Il y a une citation qui résume bien cela, prononcée par un certain Terje Mathisen. Si vous ne le connaissez pas, cet homme est un vieux programmeur (du temps durant lequel on codait encore en assembleur), grand gourou de l'optimisation, qui a notamment travaillé sur le moteur de Quake 3 Arena.

Citation : Terje Mathisen

> almost all programming can be viewed as an exercise in caching

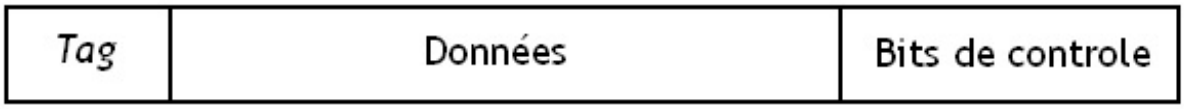
Programmeurs, faites de cette citation votre manière de programmer !

Correspondance Index - Adresse

Pus haut, j'ai dit que notre cache était capable de faire la correspondance entre un index et une adresse mémoire. Maintenant, on va voir que cette correspondance dépend beaucoup du cache, et on va aussi voir comment notre cache peut retrouver les données qu'on a stockées dedans.

Tag d'une ligne de cache

Pour faire la correspondance entre une ligne de cache et l'adresse mémoire correspondante, on utilise une petite bidouille un peu compliquée. On ajoute des bits supplémentaires à notre ligne de cache qui contiennent une partie (voir la totalité) des bits de l'adresse mémoire qui correspond à notre ligne. Ces bits supplémentaires forment ce qu'on appelle le ***Tag***.



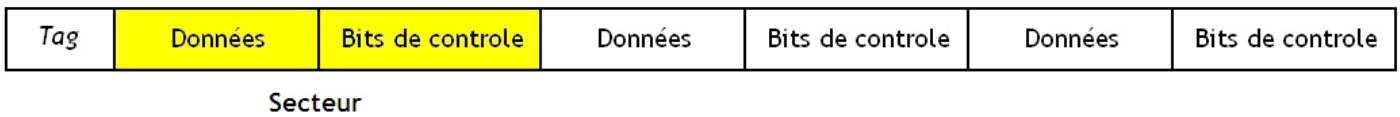
Quand notre cache reçoit une demande de lecture ou écriture, il va comparer le *Tag* de chaque ligne avec les bits de poids fort de l'adresse à lire ou écrire. Si une ligne contient ce *Tag*, alors c'est que cette ligne correspond à l'adresse demandée et un *cache hit* à lieu. Mais si aucune ligne de cache n'a ce *Tag*, alors c'est un *cache miss*.

Mine de rien, cela demande de comparer le tag avec un nombre de ligne de cache qui peut être conséquent, et qui varie suivant l'organisation du cache : certains caches n'ont pas besoin d'effectuer cette vérification du *Tag* pour toutes les lignes de caches, comme on le verra tout à l'heure.

Sector Caches

Sur certains caches assez anciens, on pouvait transférer nos lignes de caches morceaux par morceaux. Ces caches avaient des lignes de cache divisées en sous-secteurs, ces sous-secteurs étant des morceaux de ligne de cache qu'on pouvait charger indépendamment les uns des autres. On pouvait ainsi charger seulement un morceau d'une ligne de cache depuis la mémoire et ne pas charger les autres secteurs.

Sur ces caches, chaque secteur avait ses propres bits de contrôle. Mais le *Tag* était commun à tous les secteurs, ce qui fait que les morceaux de mémoire copiés dans les secteurs appartenant à une même ligne de cache appartenaient au bloc de mémoire RAM correspondant au *Tag*. Ils se suivaient en mémoire, quoi.



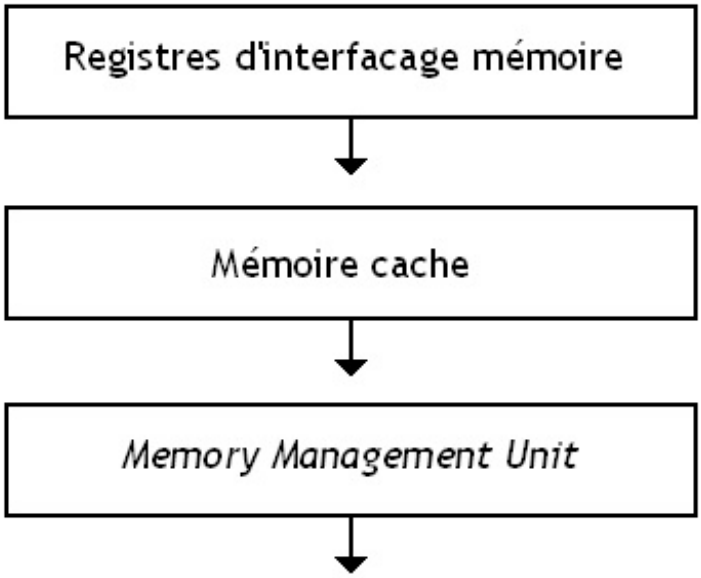
Adresses physiques ou logiques ?

Quoiqu'il en soit, on peut se poser une question : ces adresses mémoires dont on parle depuis tout à l'heure, ce sont des adresses physiques ou des adresses virtuelles ?

Et bien ça dépend du cache !

Virtually Tagged

Sur certains caches, la correspondance se fait entre une adresse virtuelle et une ligne de cache. Notre cache peut donc être placé avant la MMU dans notre processeur.



Ce genre de cache n'a donc pas besoin d'attendre que la MMU aie finie de traduire l'adresse logique en adresse physique pour vérifier si la donnée à laquelle accéder est présente ou non dans le cache : ces caches sont plus rapides.

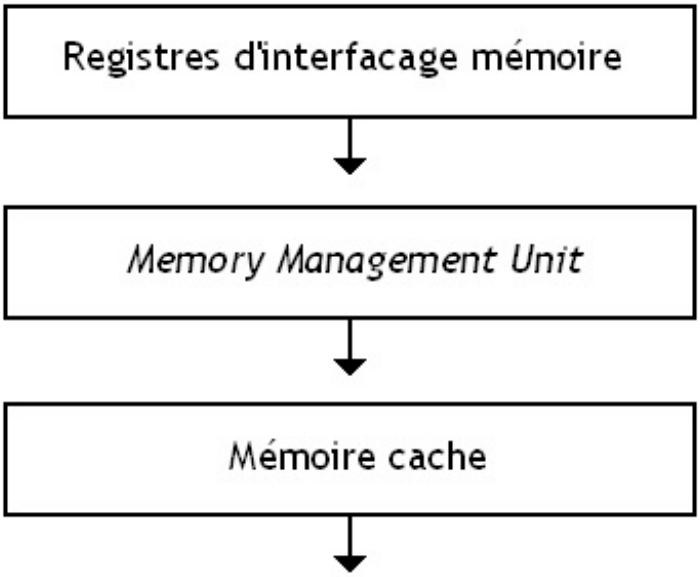
Mais les problèmes arrivent facilement quand on utilise plusieurs programmes avec ce genre de cache. En effet, plusieurs programmes peuvent utiliser la même adresse logique : pour eux, toute la mémoire est strictement libre, sans aucun programme qui viendrait prendre des adresses. Ils sont donc autorisés à utiliser toutes les adresses logiques depuis l'adresse 0, et rien ne dit qu'un autre programme n'aura pas choisit la même adresse logique pour autre chose. Cela ne pose pas problème en temps normal, vu que la même adresse logique peut correspondre à plusieurs adresses physiques suivant le programme (on dit merci à la MMU).

Mais avec notre cache, comment savoir à quel programme appartient la ligne de cache qui correspond à une adresse virtuelle ?

On n'a pas vraiment le choix : soit on rajoute dans notre ligne de cache des bits de contrôle qui permettront d'identifier le programme qui possède la ligne de cache ; soit on vide totalement le cache en changeant de programme.

Physically Tagged

Sur d'autres caches, c'est l'adresse physique qui est utilisée pour faire la correspondance entre une ligne de cache et une adresse mémoire. Notre cache doit donc être placé après la MMU dans notre processeur.

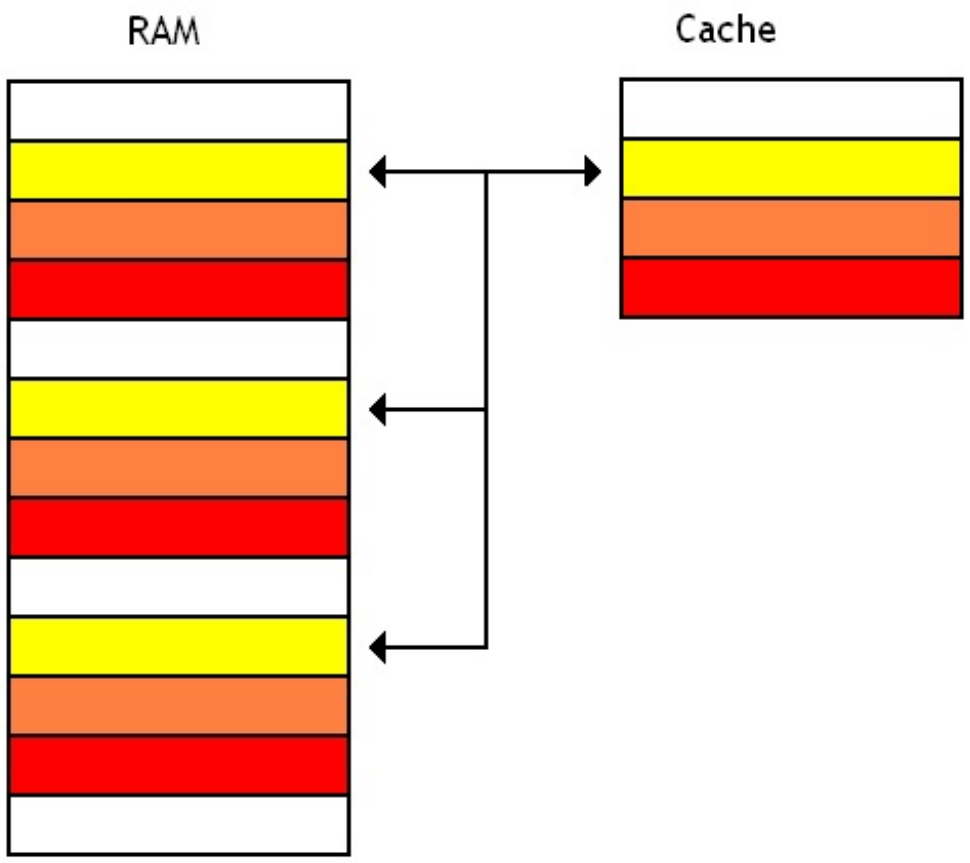


Ce genre de caches est plus lent en terme de temps d'accès : il faut attendre que la MMU traduise l'adresse logique en adresse physique pour qu'on puisse enfin demander au cache si la donnée à laquelle accéder est dans le cache ou pas. Par contre, on n'a strictement rien à faire pour partager le cache entre plusieurs programmes.

Les caches *direct mapped*

J'ai dit plus haut que certains caches n'avaient pas besoin de faire un grand nombre de comparaisons pour vérifier qu'une ligne de cache correspondait bien à une adresse accédée en lecture ou écriture. Et bien il est temps que je vous explique comment réaliser une prouesse pareille. Dans ce qui va suivre, on va comment sont organisés nos lignes de cache et comment les faire correspondre avec une adresse mémoire. Il faut savoir que cela varie suivant le cache et que divers types de caches existent. On va tout d'abord voir le cas des caches *direct mapped*.

Avec ce genre de cache, le contenu d'une adresse mémoire sera chargée dans une ligne de cache prédéfinie et impossible à modifier : ce sera toujours la même, quelques soient les circonstances.



L'accès au cache est très rapide vu qu'il suffit de vérifier une seule ligne de cache : celle qui est prédéfinie pour l'adresse à laquelle on souhaite accéder. Si le *Tag* est le bon, c'est un *cache hit*, sinon, c'est un *cache miss*.

Les circuits chargés de gérer un cache *direct mapped* sont particulièrement simples : pas besoin de faire une recherche dans tout le cache lors de l'accès à un élément (un seul circuit comparateur peut suffire), pas besoin de beaucoup "réfléchir" pour savoir où charger une donnée dans le cache, etc. En conséquence, ces caches ont un temps d'accès très faible, et chauffent très peu.

Conflit miss

Juste une remarque : le cache est plus petit que la mémoire, ce qui fait que certaines adresses mémoires devront se partager la même ligne de cache.

Si on a besoin de manipuler fréquemment des données qui se partagent la même ligne de cache, chaque accès à une donnée supprimera l'autre du cache. Et tout accès à l'ancienne donnée, qui a quitté le cache, se soldera par un *cache miss*. Ce genre de cache miss du au fait que deux adresses mémoires ne peuvent utiliser que la même ligne de cache s'appelle un ***conflit miss***.

Cela fait que le hit ratio de ce genre de cache est quelque peu...comique ! Par contre, les circuits de gestion de ce genre de cache nécessitent très peu de transistors et l'accès aux données présentes dans le cache est très rapide.

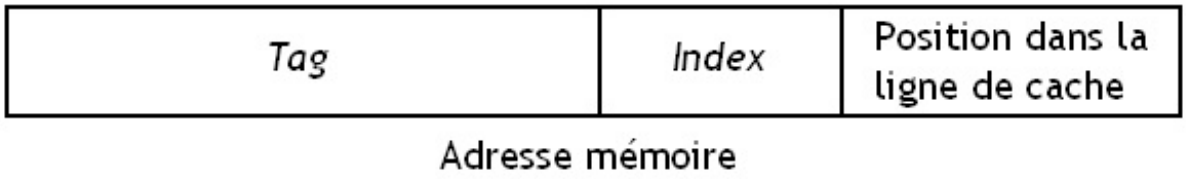
Adresse

Généralement, les concepteurs de mémoire cache s'arrangent pour que des adresses consécutives en mémoire RAM occupent des lignes de cache consécutives. Cela permet de simplifier fortement les circuits de gestion du cache, sans compter que c'est plus logique.

Sur les caches *direct mapped*, chacune de ces lignes de cache possède un **index**, un nombre entier positif ou nul qui permet de l'identifier et la sélectionner parmi toutes les autres lignes. Il ne s'agit pas d'une adresse ! Une ligne de cache n'a pas d'adresse et n'est pas adressable via le bus d'adresse : l'index qui va identifier cette ligne reste confinée dans les circuits qui s'occupent de gérer notre mémoire cache sans jamais quitter notre processeur. Notre cache n'est pas non plus mappé dans la mémoire RAM, pas plus qu'il n'existe un espace d'adressage séparé pour le cache : aucune adresse ne permet de sélectionner une ligne de cache, c'est le processeur qui gère celui-ci en interne et utilise une sorte d' "adresse interne" pour sélectionner nos lignes de cache: l'index.

Avec cette implémentation, notre adresse mémoire peut donc permettre de spécifier l'*index* de notre donnée. Le *Tag* correspond

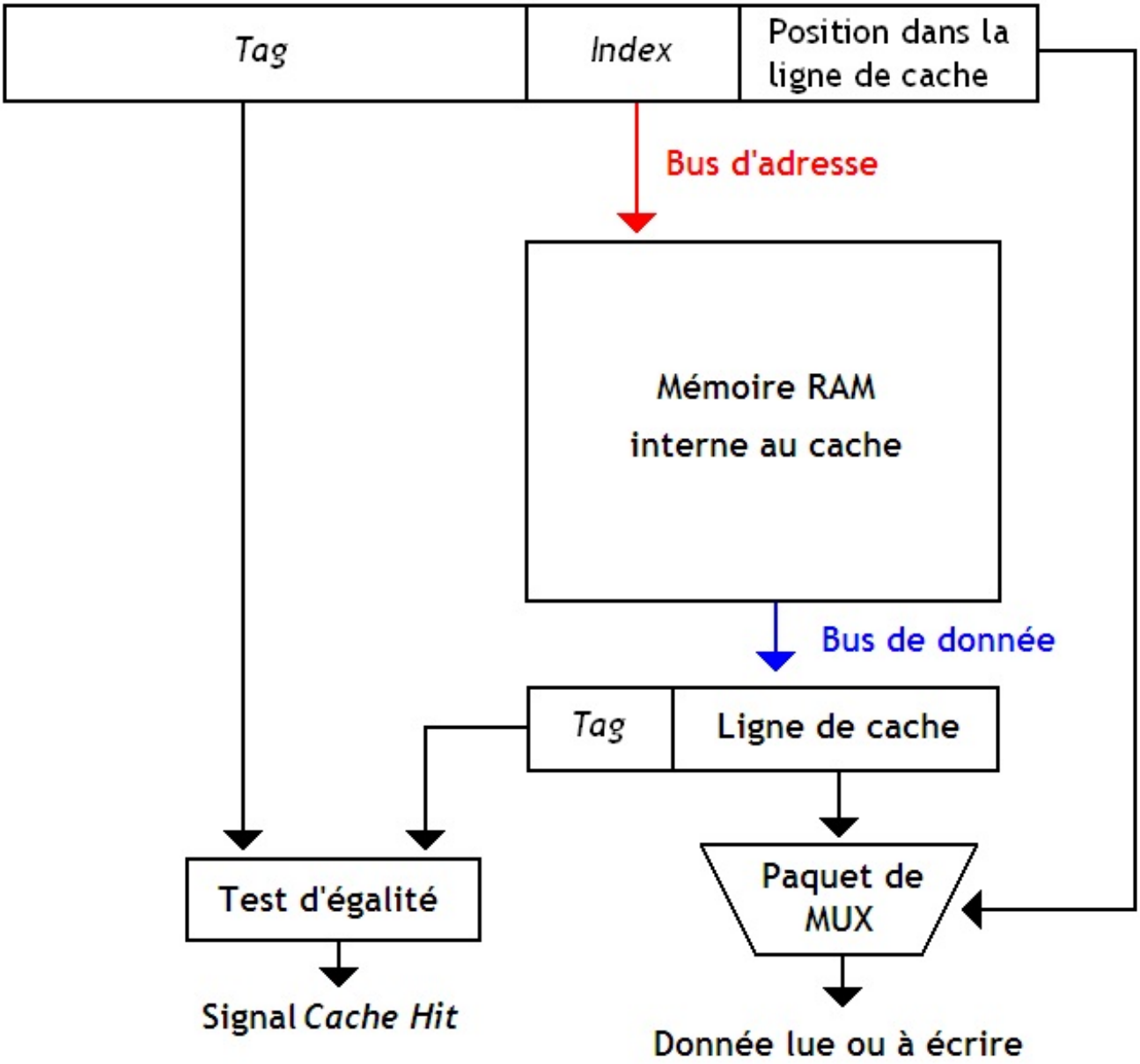
aux bits de poids forts de l'adresse mémoire correspondant à notre ligne de cache. Le reste des bits de l'adresse représente l'index de notre ligne, et la position de la donnée dans la ligne.



Implémentation

Maintenant, passons aux circuits de notre cache *Direct Mapped*. Celui-ci est conçu d'une façon assez simple : il suffit d'utiliser une RAM, un comparateur, et un paquet de multiplexeurs. La mémoire RAM servira à stocker à la fois les lignes de caches, mais aussi les *Tags* qui leur correspondent. Un *byte* de cette mémoire contiendra ainsi une ligne de cache, avec son *Tag*. Chaque ligne étant sélectionnée par son Index, on devine aisément que l'Index de notre ligne de cache sera envoyée sur le bus d'adresse de notre mémoire RAM pour sélectionner celle-ci.

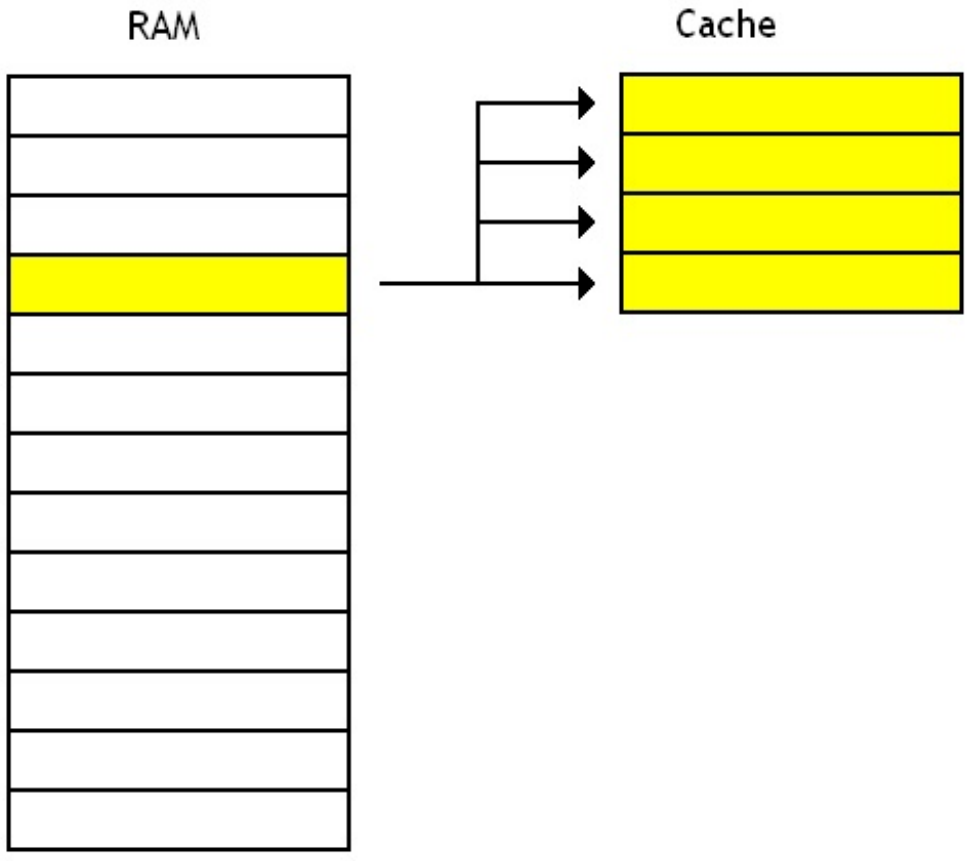
Ensuite, il suffira de comparer le *Tag* de la ligne de cache sélectionnée par l'*Index*, et le comparer avec le *Tag* de l'adresse de la donnée à lire ou écrire. On saura alors si on doit faire face à un *Cache Miss* ou si on a un joli petit *Cache Hit*. Ensuite, on devra sélectionner la bonne donnée dans notre ligne de cache avec un ensemble de multiplexeurs. Et voilà, nous obtenons un beau cache *Direct Mapped*.



On pourrait éventuellement placer les Tags et les lignes de caches dans deux mémoires RAM à part, mais c'est un détail.

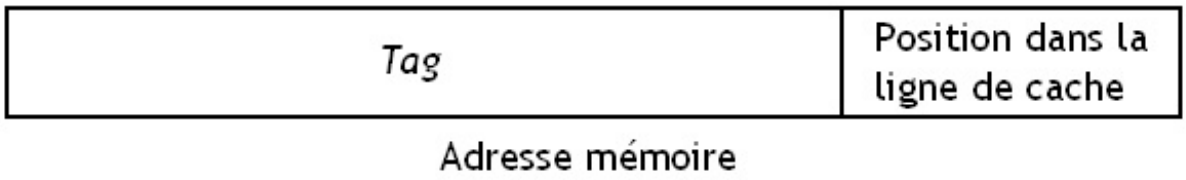
Les caches *Fully associative*

Avec les caches ***Fully Associative***, toute donnée chargée depuis la mémoire peut être placée dans n'importe quelle ligne de cache, sans aucune restriction.



Adresse

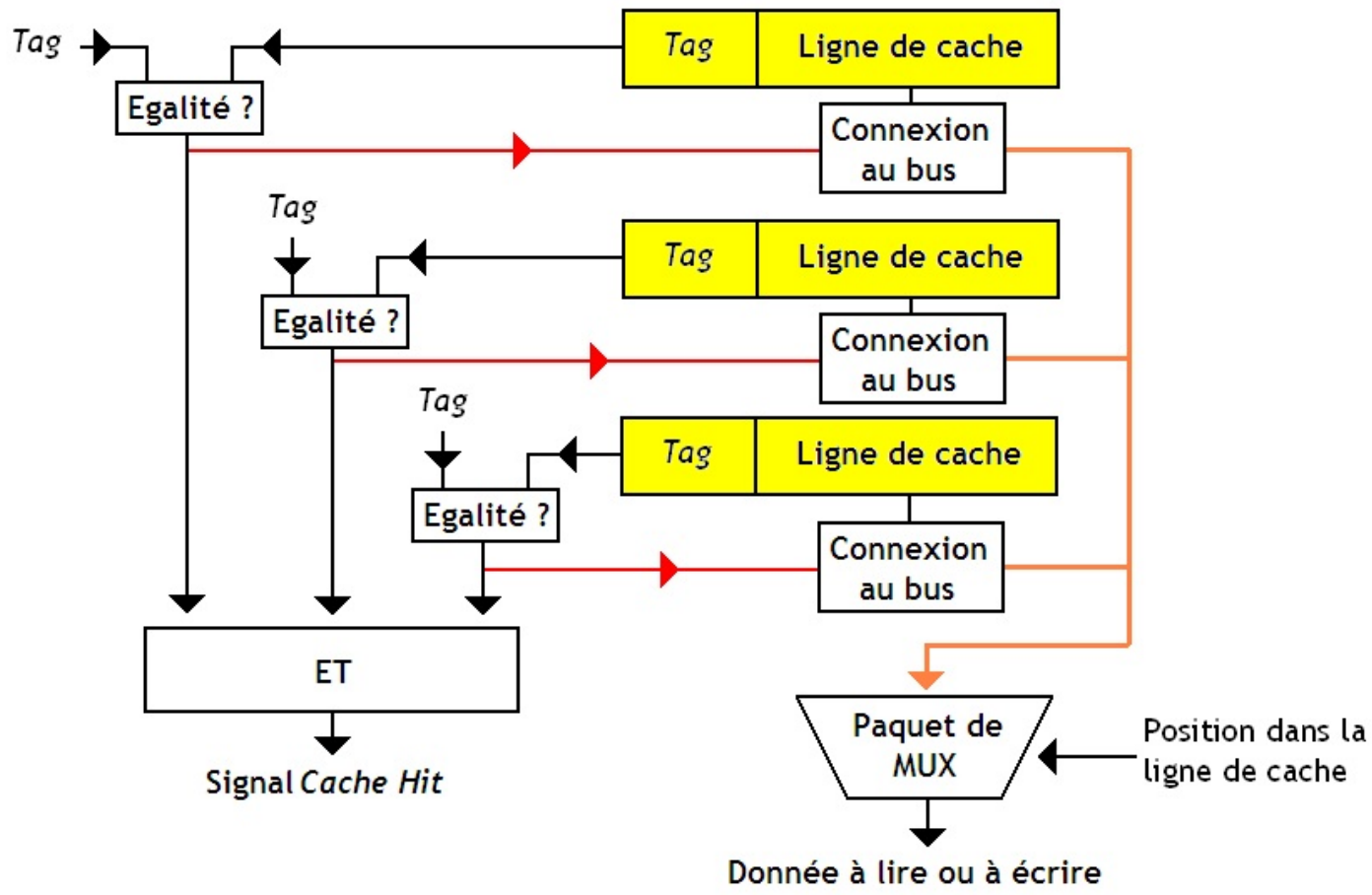
En clair, l'adresse mémoire ne peut pas servir à identifier une ligne en particulier : le format de l'adresse ne contient pas de quoi spécifier l'*Index*. Elle est donc découpée en un *Tag*, et de quoi identifier la position de la donnée dans la ligne de cache correspondante.



Ces caches ont un hit ratio très élevé, vu qu'il y a pas de possibilité d'avoir le moindre *conflit miss* : deux données différentes ne peuvent pas avoir l'obligation de se partager la même ligne, vu qu'on peut placer nos données n'importe où.

Implémentation

Concevoir des caches totalement associatif est beaucoup plus complexe que concevoir des caches *Direct Mapped*. Voolà à quoi ressemble d'organisation générale d'un cache *Fully Associative* :



D'abord, on doit déterminer si on a un *Cache Miss* ou un *Cache Hit*. Vu que notre donnée peut être placée dans n'importe quelle ligne de cache, il faut vérifier les Tags de toutes les lignes de cache. Chaque *Tag* est donc relié à un comparateur, qui vérifiera l'égalité entre le *Tag* de la ligne de cache, et le *Tag* extrait de l'adresse accédée en lecture ou en écriture. Si un seul de ces comparateur renvoie "Vrai" (en clair, 1), alors on a un *Cache Hit*. Sinon, c'est un *Cache Miss*. Le signal qui indique l'apparition d'un *Cache Hit* est donc un gros ET logique entre toutes les sorties des comparateurs.

Toutes nos lignes de caches sont reliées à un bus interne qui permettra de relier nos lignes de cache à l'extérieur. Si les deux *Tags* sont identiques, cela signifie que la ligne de cache associée est la bonne, et que c'est celle-ci qu'on doit lire ou écrire. Il faut donc la connecter à ce bus, et déconnecter les autres. Pour cela, on relie donc la sortie du comparateur à des transistors chargés de connecter ou de connecter les lignes de cache sur notre bus.

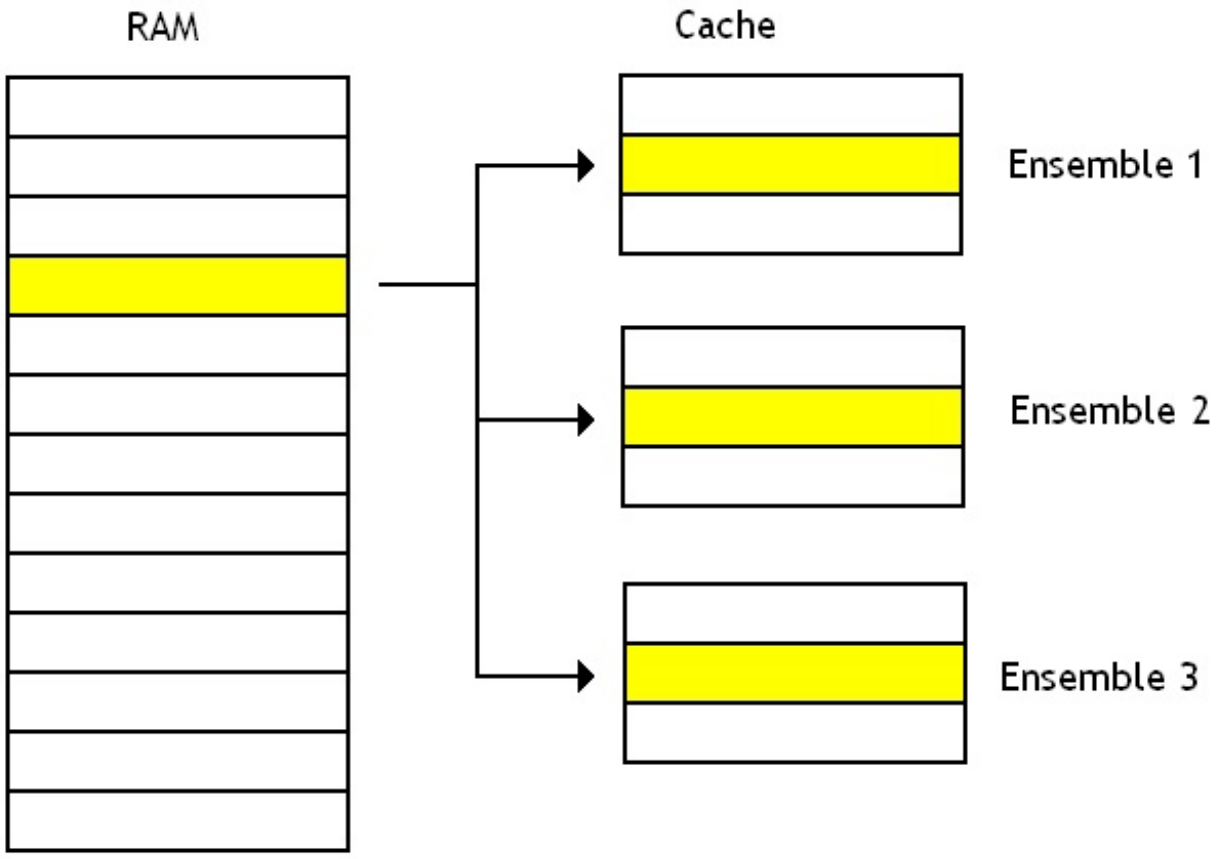
Ensuite, il ne reste plus qu'à sélectionner la portion de la ligne de cache qui nous intéresse, grâce à un paquet de multiplexeurs, comme pour les caches *Direct Mapped*.

Comme vous le voyez, ce genre de cache a besoin d'un grand nombre de comparateurs : un par ligne de cache, comparé aux caches *Direct Mapped*, qui n'ont besoin que d'un seul comparateur.
. Ça nécessite énormément de transistors et ça chauffe !

Les caches *Set associative*

Nos caches *Direct Mapped* et *Fully Associative* ont chacun leurs défaut : un *hit ratio* médiocre pour les premiers, et un temps d'accès trop long pour les autres. Les deux influent énormément sur les performances et certains caches implémentent une sorte de compromis destiné à trouver un juste milieu : ce sont les caches ***Set associative***.

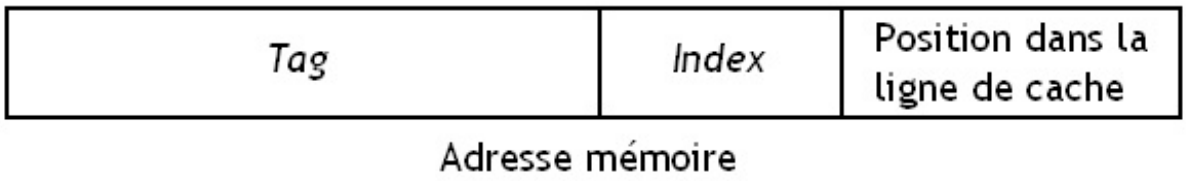
Pour simplifier, ces caches sont composés de plusieurs ensembles de lignes de caches. On peut décider ce choisir dans quelle bloc une donnée peut être chargée, mais pas la ligne de cache exacte dans laquelle charger notre donnée : les lignes de cache à l'intérieur d'un ensemble sont associées à des adresses mémoires en utilisant la stratégie *Direct Mapped*.



En clair, il s'agit d'un regroupement de plusieurs caches *Direct Mapped*, avec comme particularité le fait qu'on peut choisir dans quel cache stocker notre donnée.

Adresse

L'adresse d'une case mémoire va devoir être découpée en trois partie : un *Tag*, un *Index*, et un *Offset*. Rien ne change comparé aux caches *Direct Mapped*.

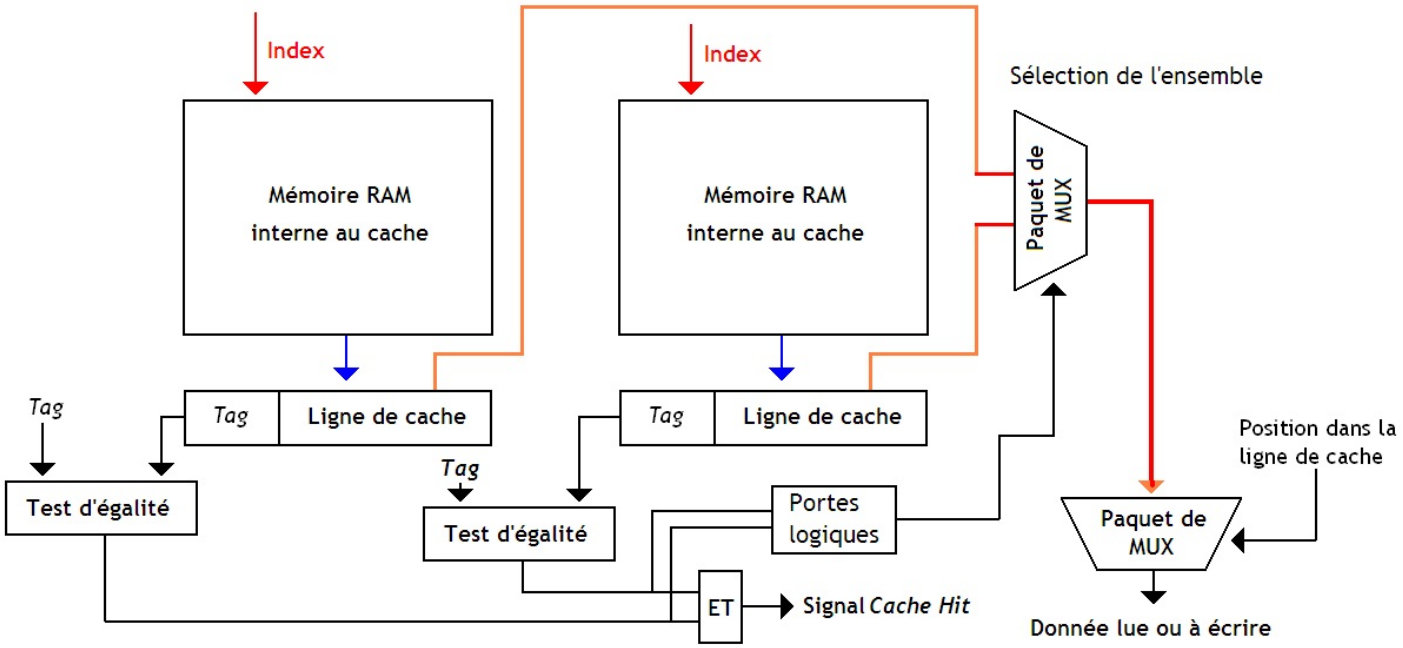


Conflit miss : le retour

Bous aurez remarqués que dans un ensemble de ligne de cache, nos lignes sont accédées en *Direct mapped* : deux données peuvent se partager la même ligne de cache si on les charge dans le même ensemble. Bien sûr, ces cas sont beaucoup plus rares qu'en *Direct mapped* : il suffit simplement de placer nos deux données dans deux ensembles différents pour éviter tout conflit. Mais cela n'est pas toujours possible, et dans certains cas rares, les *conflits miss* sont possibles sur un cache *Set associative*.

Implémentation

Comme je l'ai dit, un cache *Set Associative* peut être vu comme un regroupement de plusieurs caches *Direct Mapped*. Et bien cela se voit dans la façon dont ces caches sont conçus : leur organisation est un vrai mélange de *Direct Mapped* et de *Fully Associative*. Jugez par vous-même !



Comme vous pouvez le voir, l'organisation est identique à celle d'un cache *Fully Associative*, à part que chaque ensemble Tag-Ligne de cache est remplacé par une mémoire RAM qui en contient plusieurs. Et pour sélectionner une donnée dans cette RAM, il suffit d'envoyer son *Index* dans le bus d'adresse de toutes ces mémoires RAM, et de récupérer les couples Tag-Ligne de cache. On vérifie alors les *Tags* de ces couples, et si jamais il y a correspondance avec le *Tag* de l'adresse à lire ou écrire, on connecte la ligne de cache (celle reçue en sortie de notre mémoire RAM) sur le bus relié au MUX. Et ensuite, le MUX sélectionne la donnée voulue dans notre ligne.

Way Prediction

Les caches Set Associative sont donc une sorte de compromis entre Direct Mapped et Fully Associative. Ils ont un Hit Ratio et un temps d'accès intermédiaire. La différence de temps d'accès entre un cache Set Associative et un cache Direct Mapped vient du fait qu'on doit sélectionner l'ensemble qui contient la bonne donnée. On doit rajouter un multiplexeur pour cela. Ce qui fait que l'on prend un peu plus de temps comparé à un cache Direct Mapped.

Pour réduire ce temps d'accès et le rendre le plus similaire possible au temps d'accès qu'on pourrait avoir avec un cache Direct Mapped, certains chercheurs ont eu une idée lumineuse : la ***Way Prediction***. Cette technique consiste à faire des paris sur l'ensemble dans lequel on va aller lire une donnée. Au lieu d'attendre que les comparaisons des Tags soient effectuées, le processeur va sélectionner automatiquement un ensemble et va configurer à l'avance les multiplexeurs. Le processeur va donc parier sur un ensemble en espérant que ce soit le bon, et en espérant que l'opération finisse par un Cache Hit. Si le processeur ne se trompe pas, le processeur va accéder à la donnée de façon précoce, et commencer à l'utiliser un à deux cycles plus tôt que prévu. S'il se trompe, le processeur devra annuler la lecture effectuée en avance, et recommencer en allant chercher la donnée dans le bon ensemble.

Cette technique peut aussi avoir un autre avantage : elle peut être adaptée de façon à diminuer la consommation énergétique du processeur. Pour cela, il suffit de mettre en veille tous les caches Direct Mapped sur lesquels le processeur n'a pas parié. C'est plus efficace que d'aller lire plusieurs données dans des mémoires différentes, et n'en garder qu'une.

Reste à implémenter cette technique. Pour cela, rien de plus simple. Déjà, prédire quelle sera l'ensemble est d'une simplicité déconcertante. En vertu du principe de localité, on peut décemment penser que si on accédé à un ensemble, alors les accès futurs auront lieu dans le même ensemble. Il nous suffit donc de retenir l'ensemble le plus récemment accédé, et le tour est joué. Pour cela, il suffit de placer un registre sur l'entrée de commande du multiplexeur qui choisit l'ensemble à lire. Pour vérifier que la prédiction est correcte, il suffit de comparer le contenu de ce registre au résultat renvoyé après vérification des Tags.

Remplacement des lignes de cache

Comme on l'a dit plus haut, avoir un cache, c'est bien. Savoir quelles données charger dedans, c'est mieux ! Et là, c'est la catastrophe ! Une mauvaise utilisation du cache peut parfaitement le rendre inutile. Seul problème, le cache est une mémoire très petite, et il vaut mieux faire en sorte qu'un maximum de données utiles se trouve dans ce cache, sans gaspiller son précieux espace vital avec des données qui seront inutiles.
Il faut donc :

- choisir le mieux possible les données à charger dans le cache : cela permet d'éviter le plus possible que le processeur doive aller chercher en RAM une donnée qui n'est pas présente dans le cache ;
- supprimer les données devenues inutiles pour laisser la place à des données utiles.

Le choix des données à charger dans la mémoire est géré automatiquement par le cache et des circuits dédiés. Généralement, une donnée est chargée dans le cache quand on doit lire ou écrire dedans. Mais sur certaines processeurs, le cache est relié à un circuit qui va se charger de précharger à l'avance certaines données dont il prédit qu'elles seront utiles dans le futur. Ce circuit s'appelle le *prefetcher*, et il possède des algorithmes parfois assez sophistiqués capable de déduire quelles données seront accédées dans un futur proche.

Remplacement des lignes de cache

Vient ensuite le choix des données à garder dans le cache et le choix des données qui doivent quitter le cache pour être enregistrées dans la mémoire RAM. Lorsqu'une mémoire cache est intégralement remplie et qu'on doit charger une donnée dans ce cache (en effectuant une lecture en mémoire RAM), il faut que certaines données présentes dans une ligne de cache laissent la place pour les nouvelles données.

Dans le cas d'un cache *Direct Mapped*, il n'y a pas grand chose à faire : la donnée a déjà une position préétablie dans le cache, et on sait d'avance quelle donnée rapatrier en mémoire RAM. Mais pour les mémoires caches Fully Associative ou Set Associatives, c'est autre chose : la donnée chargée en mémoire peut prendre plusieurs place différentes dans la mémoire. Dans une mémoire Fully Associative, la donnée peut aller n'importe où. Et dans un cache Set Associatives, la donnée peut prendre N places différentes : une dans chaque sous-cache Direct Mapped. Il faut donc décider où placer notre nouvelle donnée.

Comment choisir ?

Évidemment, le choix des données à rapatrier en mémoire RAM ne peut pas être fait n'importe comment : rapatrier une donnée qui sera surement utilisée sous peu est inutile, et il vaudrait mieux supprimer des données qui ne serviront plus ou dans alors dans longtemps. Il existe différents algorithmes spécialement dédié à résoudre ce problème efficacement, directement câblés dans les unités de gestion du cache. Certains sont vraiment très complexes, aussi je vais vous présenter quelques algorithmes particulièrement simples.

Implémentation

Mais avant de voir ces algorithme, il faut absolument que je vous parle d'une chose très importante. Quelque soit l'algorithme en question, il va obligatoirement faire une chose : choisir une ligne de cache parmi toute les autres. Et une fois cette ligne de cache choisie, il va devoir recopier son contenu dans la RAM. Notre algorithme va donc devoir identifier et sélectionner cette ligne de cache parmi toute les autres. Difficile de faire ceci efficacement en utilisant nos Tags. Pour résoudre ce problème, notre circuit de sélection des lignes de cache à remplacer va pouvoir adresser chaque ligne de cache ! Et oui, vous avez bien vu : chaque ligne de cache sera numérotée par une adresse, interne au cache. Ainsi, notre algorithme de suppression des lignes de cache a juste à former l'adresse de la ligne de cache à remplacer.

Aléatoire

Premier algorithme : la donnée effacée du cache est choisie au hasard ! Si l'on dispose d'un cache avec N lignes, cet algorithme s'implémente avec un circuit qui fournit un nombre aléatoire, compris entre 0 et N. Bien sûr, cette génération de nombre aléatoire n'a pas besoin d'être des plus parfaite : le nombre tiré n'a pas besoin d'être obtenu de façon totalement aléatoire, et une simple approximation du hasard suffit largement.

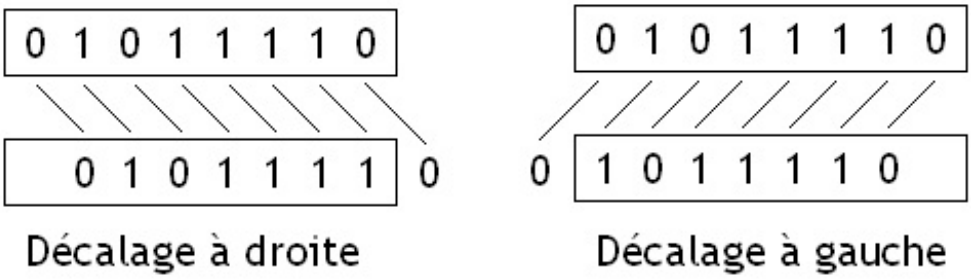
A première vue, cet algorithme ne paye pas de mine. Mais dans les faits, cet algorithme se débrouille relativement bien, et donne des résultats assez honorables. D'ailleurs, certains processeurs ARM, utilisés dans l'embarqué, utilisent cet algorithme, qui n'est clairement pas parmi les pires. Cet algorithme a aussi un avantage : son implémentation utilise très peu de portes logiques. Juste un vulgaire compteur ou un registre couplé à un circuit combinatoire très simple. Le tout prend moins d'une centaine de transistors. C'est très peu comparé aux autres algorithmes, qui prennent beaucoup plus de circuits. Autant dire qu'ils ont intérêt à être efficaces.

Clock Counter

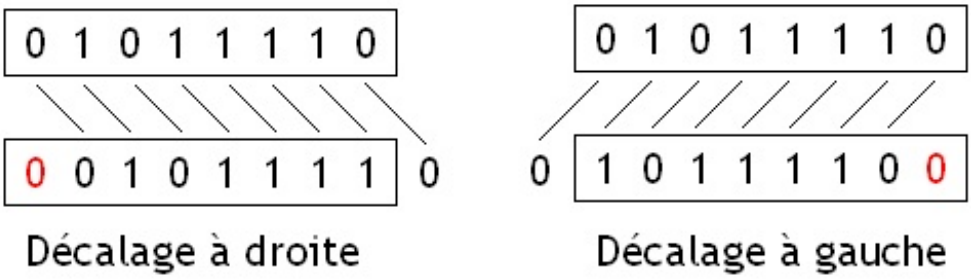
Reste à l'implémenter cet algorithme. Pour cela, on peut utiliser une première solution : un simple compteur qui s'incrémente à chaque cycle d'horloge. Généralement, les Caches Miss n'ont pas lieu à chaque cycle d'horloge, et sont souvent séparés par un nombre assez important et irrégulier de cycles d'horloge. Dans ces conditions, cette technique donne un bon résultat.

Linear Shift Register

Mais il est aussi possible d'utiliser des circuits un peu plus élaborés. L'idée consiste à utiliser des registres à décalages un peu spéciaux qu'on appelle des *Linear Shift Register*. Ces registres sont des registres à décalage. Pour rappel, un registre à décalage est un registre dont le contenu est décalé d'un cran vers la gauche (ou vers la droite) à chaque cycle d'horloge.

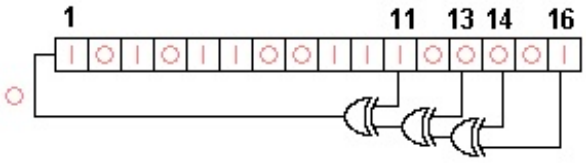


Comme vous le savez, si le contenu de ce registre est décalé d'un cran vers la gauche, le bit de poids fort de ce registre est alors mit soit à 1, soit à zéro. Sur les registres à décalages normaux, ce bit de poids fort est rempli par une valeur par défaut après chaque décalage: que ce soit un 0 ou un 1, c'est toujours le même bit qui rentre par la gauche.

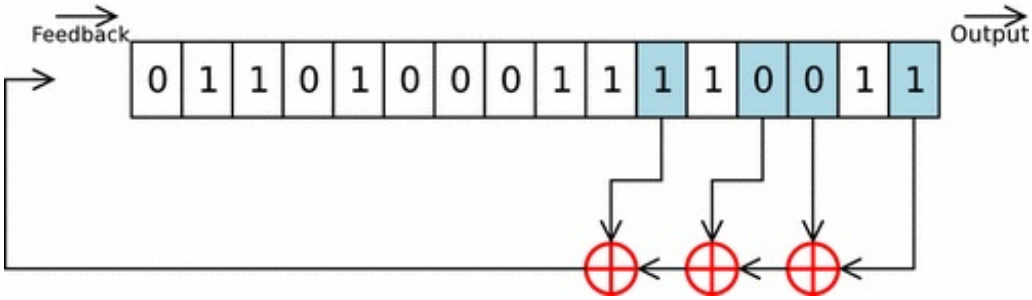


Mais dans un *Linear Shift Register*, le bit à faire rentrer à gauche change à chaque fois. Ce bit est calculé par un petit circuit combinatoire, qui est couplé au registre à décalage. Ce circuit combinatoire va prendre en entrée le contenu du registre à décalage, et va en déduire le bit à faire rentrer dedans. Ce circuit va donc utiliser tout ou partie des bits du registre à décalage, faire quelques opérations simples dessus, et en déduire quel bit faire rentrer à gauche (ou à droite).

Le contenu du registre à décalage est donc notre nombre aléatoire, qu'on utilise comme adresse de ligne de cache. Suivant la complexité du circuit combinatoire, on peut obtenir des résultats plus ou moins proches de l'aléatoire. Le circuit le plus simple consiste à utiliser quelques portes XOR sur certains bits du registre à décalage.



Le résultat donne une estimation assez faible de l'aléatoire.



Mais on peut aussi faire plus compliqué pour obtenir un bon résultat.

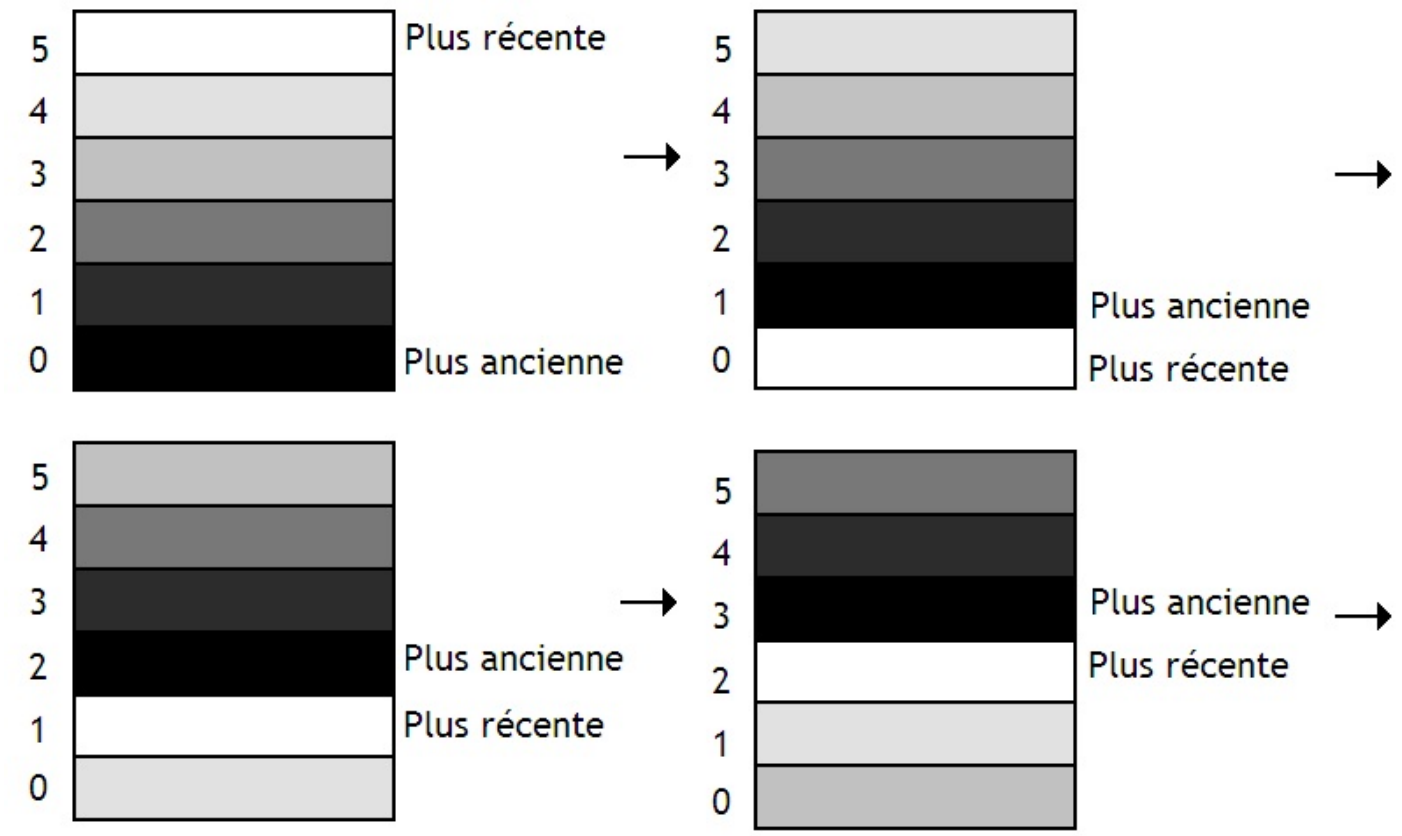
FIFO : First Input First Output

Avec l'algorithme FIFO, **la donnée effacée du cache est la plus ancienne** : c'est celle qui a été chargée dans la mémoire cache

avant les autres. Cet algorithme particulièrement stupide a une efficacité qui est souvent inférieure à l'algorithme aléatoire. Dans la majorité des cas, il y a peu de différences, mais l'avantage est tout de même du coté de l'algorithme aléatoire. En tout cas, l'algorithme FIFO est un des plus simple à implémenter en circuit : un vulgaire compteur, incrémenté lors d'un Cache Miss suffit.

Implémentation

Pour implémenter cet algorithme, la solution la plus simple permet de se contenter d'un simple compteur. Prenons un cache Fully Associative : dans celui-ci, une nouvelle donnée peut aller n'importe où dans le cache. On peut profiter de cette propriété pour insérer les données dans le cache les unes à la suite des autres. Exemple : si j'ai inséré une donnée dans la ligne de cache numéro X, alors la prochaine donnée ira dans la ligne numéro X+1. Si jamais on déborde, on revient automatiquement à zéro. En faisant ainsi, nos lignes de caches seront triées de la plus ancienne à la plus récente automatiquement.



La ligne de cache la plus ancienne sera localisée à un certain endroit, et la plus récente sera localisée juste avant. Il nous suffit juste de se souvenir de la localisation de la donnée la plus ancienne, et le tour est joué. Cela peut se faire avec un compteur unique pour tout le cache si celui-ci est Fully Asscoaitive. Pour un cache Set Associative, on doit avoir un compteur par ensemble.

Anomalie de Belady

Cet algorithme possède une petite particularité qui concerne directement les caches Set Associative. Vous pensez surement qu'en augmentant le nombre d'ensembles, les performances augmentent : on diminue le nombre de Conflicts Miss. Et bien avec un algorithme de remplacement des lignes de cache FIFO, les performances peuvent se dégrader. C'est ce qu'on appelle l'**anomalie de Belady**.

MRU : Most Recently Used

Avec l'algorithme MRU, **La donnée qui est effacée est celle qui a été utilisée le plus récemment**. Cet algorithme de remplacement est très utile quand un programme traverse des tableaux éléments par éléments. C'est quelque chose d'assez communs dans beaucoup de programmes : il n'est pas rare que nos programmes parcours des tableaux du premier élément jusqu'au dernier.

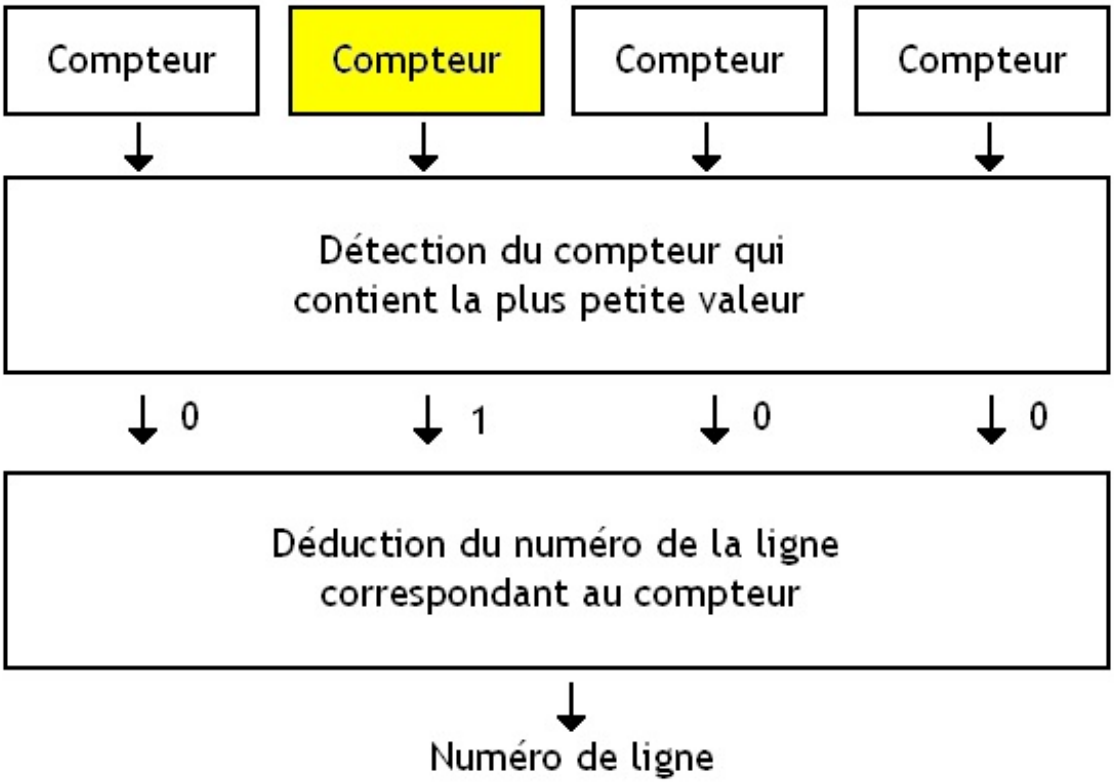
Dans ces conditions, l'utilisation des algorithmes précédents a tendance à remplacer les données du cache avec les données du tableau. Données qui ne seront souvent jamais réutilisées. En utilisant la politique MRU, notre cache ne se remplira pas avec ces données inutiles et conservera ses données utiles. Et pour être franc, il est prouvé que dans ces conditions, l'algorithme MRU est optimal. Mais dans toutes les autres conditions, cet algorithme a des performances assez misérables.

Cet algorithme s'implémente simplement avec un registre, dans lequel on place le numéro de la dernière ligne de cache utilisée.

LFU : Last Fréquently Used

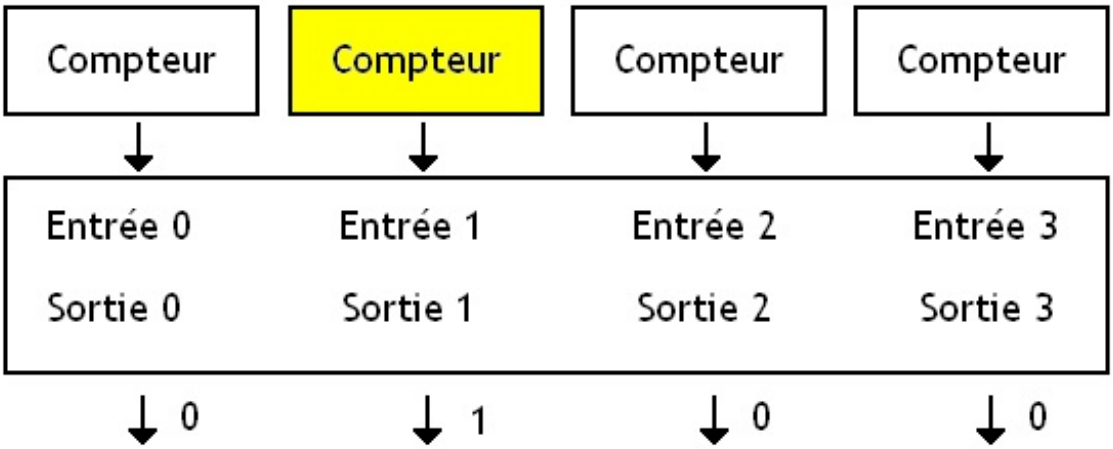
Avec l'algorithme LFU, **la donnée supprimée est celle qui est utilisée le moins fréquemment.** Cet algorithme s'implémente facilement. Il suffit d'associer un compteur à chaque ligne de cache. A chaque fois qu'on va lire ou écrire dans une ligne de cache, le compteur associé est incrémenté. La ligne la moins récemment utilisée est celle dont le compteur associé a la plus petite valeur.

On s'aperçoit rapidement qu'implémenter cet algorithme prend pas mal de transistors : il faut d'abord rajouter autant de compteurs qu'il y a de lignes de cache. Ensuite, il faut rajouter un circuit pour déduire quel compteur contient la plus petite valeur. Et enfin, il faut aussi rajouter un circuit pour déduire le numéro de la ligne de cache en fonction du compteur déterminé à l'étape précédente.



Premier circuit

Le circuit qui détermine quel compteur est le bon est assez simple : il a N entrées, numérotées de 0 à N-1. De plus, il dispose de N sorties, numérotées de 0 à N-1. Si jamais le compteur placé sur l'entrée numéro X contient la plus petite valeur, alors la sortie X est mise à 1, et toute les autres sont mises à 0.



On peut facilement créer ce circuit avec un gros paquet de comparateurs et quelques portes logiques ET. Réfléchissez un petit peu, la solution est évidente.

Second circuit

Le dernier circuit est un peu plus complexe. Celui-ci va avoir N entrées, numérotées de 0 à N-1, et une sorties. Sur cette sortie, le circuit fournira le numéro de la ligne de cache la moins fréquemment utilisée. Ce numéro est déduit en fonction de l'entrée qui a pour valeur 1.

Ce circuit est ce qu'on appelle un Encodeur, et il peut être vu comme un cousin du décodeur. Si vous regardez bien, ce circuit fait exactement l'inverse de ce que fit un décodeur : au lieu de prendre une adresse et d'en déduire la sortie à mettre à 1, il fait l'inverse : en fonction de l'entrée à 1, il en déduit une adresse. Mais attention : ne croyez pas que ces deux circuits sont conçus de la même façon. Ces deux circuits ont des câblages totalement différents.

LRU : Last Recently Used

Avec l'algorithme LRU : **la donnée remplacée est celle qui a été utilisée le moins récemment**. Cet algorithme se base sur le principe de localité temporelle, qui stipule que si une donnée a été accédée récemment, alors elle a de fortes chances d'être réutilisée dans un futur proche. Et inversement, toute donnée peu utilisée récemment a peu de chance d'être réutilisée dans le futur. D'après le principe de localité temporelle, la donnée la moins récemment utilisée du cache est donc celle qui a le plus de chance de ne servir a rien dans le futur. Autant la supprimer en priorité pour faire la place à des données potentiellement utiles.

Implémenter cet algorithme LRU peut se faire de différentes manières. Dans tous les cas, ces techniques sont basées sur un même principe : les circuits reliés au cache doivent enregistrer les accès au cache pour en déduire la ligne la moins récemment accédée. Évidemment, mémoriser l'historique des accès au cache ne se fait pas sans matériel adapté, matériel qui prendra de la place et des transistors.

Approximations du LRU

Comme on l'a vu, implémenter le LRU coute cher en transistors. Bien plus qu'implémenter un algorithme FIFO ou aléatoire. Le choix de la ligne de cache la moins récemment utilisée demande des circuits contenant beaucoup de transistors. Plus précisément, le nombre de transistors est proportionnel au carré du nombre de lignes de cache. Autant dire que le LRU devient impraticable sur de gros caches. Le LRU a un défaut : trouver la ligne de cache la moins récemment utilisée prend pas mal de temps, et nécessite des circuits qui bouffent du transistor par pelletés de 12. C'est dommage, car le LRU est un algorithme particulièrement efficace.

Pour résoudre ce problème, nos processeurs implémentent des variantes du LRU, moins couteuses en transistors, mais qui ne sont pas exactement du LRU : ils donnent un résultat assez semblable au LRU, mais un peu plus approximatif. En clair, ils ne sélectionnent pas toujours la ligne de cache la moins récemment utilisée, mais une ligne de cache parmi les moins récemment utilisée.

Il faut dire que les lignes les moins récemment utilisées ont toutes assez peu de chance d'être utilisées dans le futur. Entre choisir de remplacer une ligne qui a 0,5% de chance d'être utilisée dans le futur et une autre qui a une chance de seulement 1%, la différence est négligeable : cela aura une influence assez faible en terme de Hit Ratio. Mais les gains en terme de circuit ou de temps d'accès au cache de ces algorithmes peuvent donner des résultats impressionnants.

Splitted LRU

L'algorithme le plus simple consiste à couper le cache (ou chaque ensemble du cache s'il est Set Associative) en deux. L'algorithme consiste à choisir le morceau de cache le moins récemment utilisé, et de choisir aléatoirement une ligne de cache dans ce morceau. Pour implémenter cet algorithme, il nous suffit d'une simple bascule qui mémorise le morceau la moins récemment utilisé, et d'un circuit qui choisit aléatoirement une ligne de cache dans ce morceau.

PLRUm

Autre algorithme, un peu plus efficace : le Pseudo LRU de type m. Cet algorithme est assez simple : à chaque ligne de cache, on attribue un bit. Ce bit sert à indiquer de façon approximative si la ligne de cache associée est une candidate pour un remplacement ou non. Si ce bit est à 1, cela veut dire : attention, cette ligne n'est pas une candidate pour un remplacement. Inversement, si ce bit est à zéro, la ligne peut potentiellement être choisie pour laisser la place aux jeunes.

Lorsque l'on lit ou écrit dans une ligne de cache, ce bit est mit à 1. Histoire de dire : pas touche ! Évidemment, au fil du temps, toutes les lignes de cache finiront par avoir leur bit à 1. Aussi, l'algorithme permet de remettre les pendules à l'heure. Si tous les bits sont à 1, on les remet tous à zéro, sauf pour la dernière ligne de cache accédée.

L'idée derrière cet algorithme est d'encercler la ligne de cache la moins récemment utilisée au fur et à mesure des accès. Tout commence lorsque l'on remet tous les bits associés aux lignes de cache à 0 (sauf pour la ligne accédée en dernier). Puis, au fur et à mesure que nos lignes de cache voient leurs bits passer à un, l'étau se ressert autour de notre ligne de cache la moins utilisée. Et on finit par l'encercler de plus en plus : au final, après quelques accès, l'algorithme donne une estimation particulièrement fiable. Et comme les remplacement de lignes de caches sont rares comparés aux accès aux lignes, cet algorithme finit par donner une bonne estimation avant qu'on aie besoin d'effectuer un remplacement.

LRU amélioré

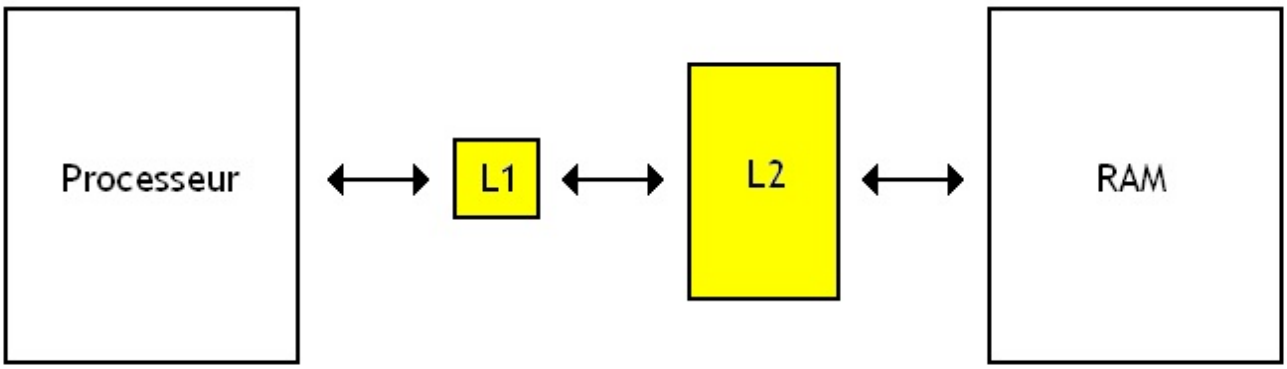
L'algorithme LRU, ainsi que ses variantes approximatives, sont très efficaces dans la majorité des programmes. Du moment que notre programme respecte relativement bien la localité temporelle, cet algorithme donnera de très bons résultats : le Hit Ratio du cache sera très élevé. Par contre, cet algorithme se comporte assez mal dans certaines circonstances, et notamment quand on traverse des tableaux. Dans ces conditions, on n'a pas la moindre localité temporelle, mais une bonne localité spatiale. Pour résoudre ce petit inconvénient, des variantes du LRU existent : celles-ci combinent plusieurs algorithmes à la fois et vont choisir lequel de ces algorithmes est le plus adapté à la situation. Notre cache pourra ainsi détecter si il vaut mieux utiliser du MRU, du LRU, ou du LFU suivant la situation.

On n'a pas qu'un seul cache !

On pourrait croire qu'un seul gros cache est largement suffisant pour compenser la lenteur de la mémoire. Hélas, nos processeurs sont devenus tellement rapides que nos caches sont eux mêmes trop lents ! Pour rappel, plus une mémoire peut contenir de données, plus elle est lente. Et nos caches ne sont pas épargnés. Après tout, qui dit plus de cases mémoires à adresser dans un cache dit décodeur ayant plus de portes logiques : les temps de propagation qui s'additionnent et cela fini par se sentir. Si on devait utiliser un seul gros cache, celui-ci serait beaucoup trop lent. La situation qu'on cherche à éviter avec la mémoire principale (la RAM) revient de plus belle. Même problème, même solution : si on a décidé de diviser la mémoire principale en plusieurs mémoires de taille et de vitesse différentes, on peut bien faire la même chose avec la mémoire cache.

Caches L1, L2 et L3

Depuis environ une vingtaine d'années, nos caches sont segmentés en plusieurs sous-caches : les caches L1, L2 et parfois un cache L3. Certains de ces caches sont petits, mais très rapides : c'est ceux auxquels on va accéder en priorité. Viennent ensuite d'autres caches, de taille variables, mais plus lents.

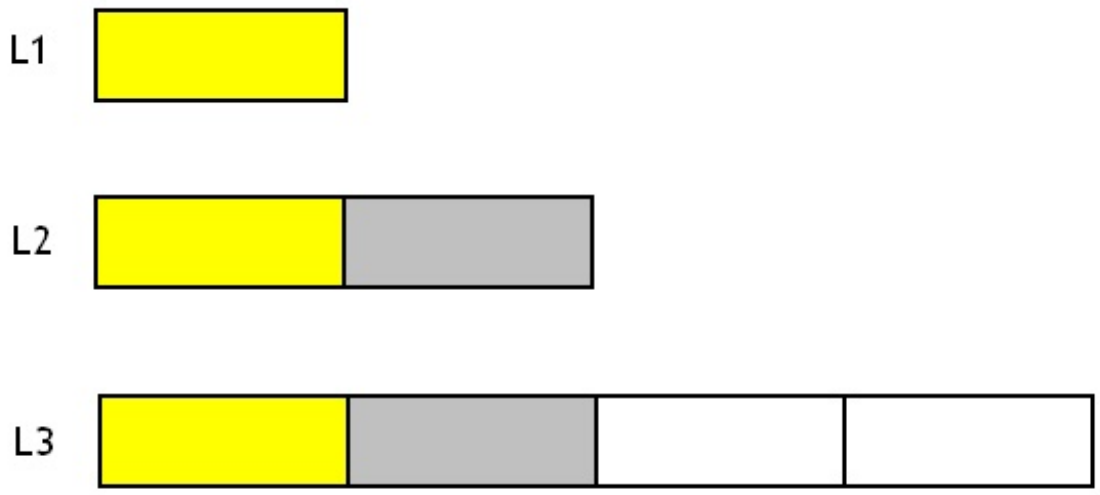


Le L1 est le cache le plus petit et le plus rapide. C'est celui dans lequel le processeur va d'abord chercher les données/instructions voulues. On trouve ensuite un cache L2, de vitesse et de taille moyenne, et parfois un cache L3, assez lent mais assez gros.

L'accès au cache est simple : on commence par vérifier si notre adresse correspond à une donnée du cache le plus rapide (qui est souvent le plus petit) : j'ai nommé le cache L1. Si elle ne l'est pas , on effectue la même vérification pour le cache de niveau inférieur (le L2). Si une donnée est présente dans un des caches, on la charge directement dans le séquenceur (instruction) ou en entrée des unités de calcul (donnée). Dans le cas contraire, on vérifie le cache du niveau inférieur. Et on répète cette opération, jusqu'à avoir vérifié tous les caches. : on doit alors aller chercher la donnée en mémoire.

Caches inclusifs

Ces caches sont organisés différemment, et leur contenu varie suivant le cache. Dans le cas des caches inclusifs, le contenu d'un cache est recopié dans les caches de niveau inférieur. Par exemple, le cache L1 est recopié dans le cache L2 et éventuellement dans le cache L3.



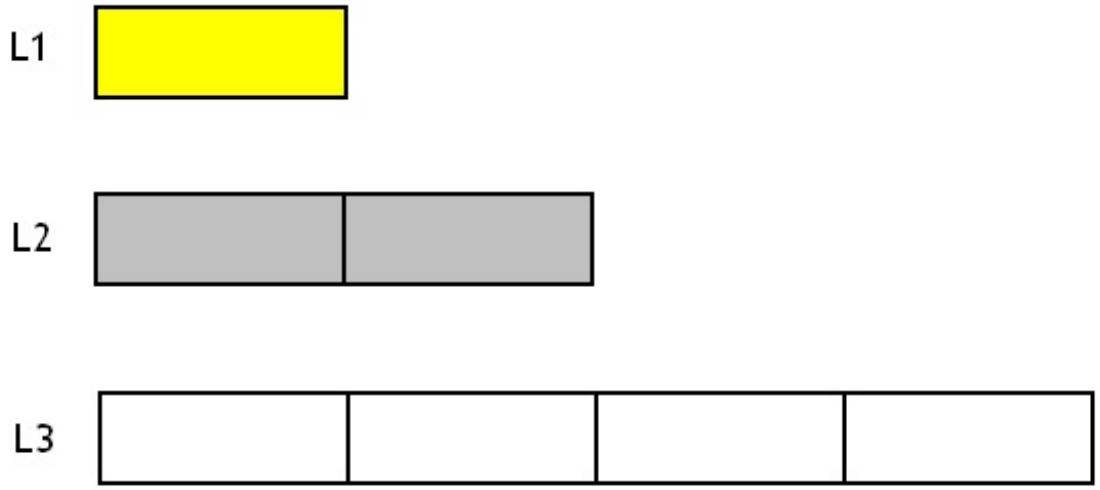
Ce genre de cache a un avantage : le temps d'accès à une donnée présente dans le cache est plus faible. Cela est du en partie à la présence des copies des caches de niveau supérieurs dans les caches de niveau inférieurs (mais pas que). Pour expliquer pourquoi, prenons un exemple : si la donnée voulue n'est pas dans le cache L1, on n'est pas obligé de vérifier la partie du cache L2 qui contient la copie du L1. Ainsi, les circuits qui doivent vérifier si une adresse correspond bien à une donnée placée dans le cache peuvent être simplifiés et ne pas vérifier certaines portions du cache. Ce qui donne un résultat plus rapide.

En contrepartie, une partie des caches de niveau inférieur contient les donnes contenues dans le cache de niveau supérieur, qui sont donc en double. Exemple, le cache L2 contient des données présentes dans le L1. De même, le cache L3 contient une copie du L2. Une partie du cache est donc un peu inutilisée, ce qui fait que c'est comme si le cache était plus petit. Celui-ci pouvant contenir moins de données utiles (on ne compte pas les données présentes en double dans les différents niveaux de cache), on a parfois un plus grand risque de *cache miss*, quand le cache est très utilisé.

De plus, la mise à jour du contenu d'un niveau de cache doit être répercutée dans les niveaux de cache inférieurs et/ou supérieurs. On doit donc transférer des informations de mise à jour entre les différents niveaux de caches.

Caches exclusifs

Dans les caches exclusifs, le contenu d'un cache n'est pas recopié dans le cache de niveau inférieur. Chaque cache contient des données différentes. Ainsi, le cache ne contient pas de donnée en double et on utilise 100% de la capacité du cache. Le cache contenant plus de donnée, on a plus de chance d'avoir un *cache hit*.



Par contre, le temps d'accès à une donnée présente dans le cache est plus long : il faut vérifier l'intégralité du contenu des caches

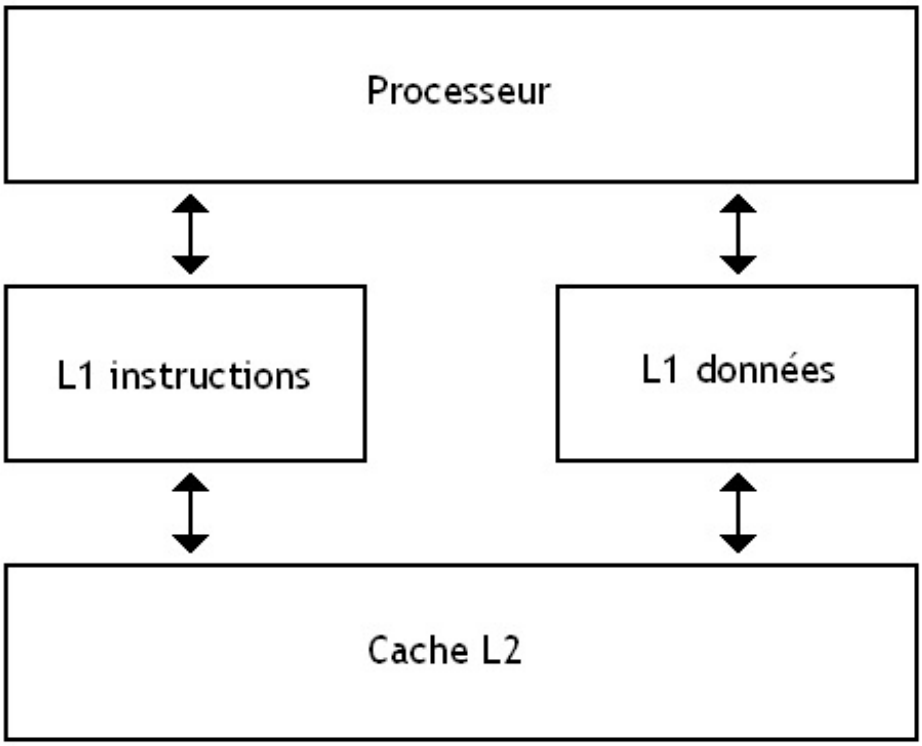
de niveau inférieur. Par exemple, si une donnée n'est pas dans le cache L1, on doit vérifier l'intégralité du cache L2, puis du cache L3. De plus, assurer qu'une donnée n'est présente que dans un seul cache nécessite aux différents niveaux de caches de communiquer entre eux pour garantir que l'on a pas de copies en trop d'une ligne de cache. Et ces communications peuvent prendre du temps.

Caches non-inclusifs et non-exclusifs

Et enfin, je me dois de parler des caches qui ne sont ni inclusifs, ni exclusifs. Sur ces caches, chaque niveau de cache gère lui-même ses données, sans se préoccuper du contenu des autres caches. Pas besoin de mettre à jours les niveaux de caches antérieurs en cas de mise à jour de son contenu, ou en cas d'éviction d'une ligne de cache. La conception de tels caches est bien plus simple.

Caches d'instruction

Au fait, sur certains processeurs, le cache L1 est segmenté en deux sous-caches : un qui contient des instructions, et un autre qui ne contient que des données.



Ces deux caches étant dans des mémoires séparées et étant reliés au reste du processeur par des bus séparés, on peut charger une instruction et manipuler une donnée en même temps, ce qui est un gros avantage en terme de performances.

Cache d'instruction et Self-Modifying Code

Le cache L1 dédié aux instructions est souvent en "lecture seule" : on ne peut pas modifier son contenu dedans, mais juste lire son contenu ou charger des instructions dedans. Cela pose parfois quelques légers problèmes quand on cherche à utiliser du *self-modifying code*, c'est à dire un programme dont certaines instructions vont aller en modifier d'autres, ce qui sert pour faire de l'optimisation ou est utilisé pour compresser voire cacher un programme (les virus informatiques utilisent beaucoup de genre de procédés). Quand le processeur exécute donc ce code, il ne peut pas écrire dans ce cache L1 d'instructions, mais va devoir écrire en mémoire cache L2 ou en RAM, et va ensuite devoir recharger les instructions modifiées dans le cache L1, ce qui prend du temps ! Et pire : cela peut parfois donner lieu à des erreurs dans certains cas. Mais bref, passons.

Decoded Instruction Cache

Sur certains processeurs, l'étape de décodage est assez complexe, voire lente. Pour accélérer cette étape, et éviter qu'elle ralentisse tout le processeur, certains concepteurs de processeurs ont décidés d'utiliser la (ou les) mémoire cache dédiée aux instructions pour accélérer ce décodage. Lorsque ces instructions sont chargées depuis la mémoire ou les niveaux de cache inférieurs, celles-ci sont partiellement décodées.

On peut par exemple rajouter des informations qui permettent de délimiter nos instructions ou déterminer leur taille : c'est très utile pour les processeurs qui utilisent des instructions de taille variable. On peut aussi détecter la présence de branchements, pour informer les divers *prefetchers*, histoire qu'ils fassent leur travail correctement. On peut aussi faire beaucoup de transformations de ce genre sur nos instructions, comme harmoniser le opcodes, etc. Bref, notre cache d'instructions peut se charger d'une partie du décodage des instructions, grâce à un petit circuit spécialisé, séparé de l'unité de décodage d'instruction.

Caches spécialisés

Il existe parfois des caches spécialisés internes au processeur, et qui ne servent pas forcément à accélérer les accès mémoire.

Loop Buffer

Pour donner un exemple, nos processeurs récents utilisent un cache spécialisé qu'on ***Loop Buffer***. Il sert à accélérer l'exécution des boucles (pour rappel, les boucles sont des structures de contrôle qui permettent de répéter une suite d'instruction). Il est chargé de stocker les µops correspondant à des instructions machines qui ont déjà étés décodées et exécutées, dans le cas où elles devraient être ré-exécutées plus tard.

Sans *Loop Buffer*, si une instruction est exécutée plusieurs fois, elle doit être chargée depuis la mémoire et décodée, à chaque fois qu'on veut l'exécuter. Le *Loop Buffer* permet d'éviter de devoir *Fetcher* et décoder cette instruction plusieurs fois de suite. Avec ce *Loop Buffer*, notre instruction est chargée et décodée en micro-opérations une seule fois, et le résultat est stocké dans ce fameux *Loop Buffer*. Si notre instruction doit être ré-exécutée, il suffit d'aller chercher le résultat du décodage de l'instruction directement dans le *Loop Buffer* au lieu de recharger l'instruction et la redécoder.

Si notre instruction est exécutée dans une boucle contenant très peu d'instructions machines, le *trace cache* va fonctionner. De même, si une instruction micro-codée est émulée par une suite de µops contenant une boucle (l'instruction REP SCACB du jeu d'instruction x86, par exemple), ce *Loop Buffer* va aider. Néanmoins, ce cache ne peut stocker qu'un nombre limité d'instruction. Si une boucle dépasse ce nombre d'instruction maximal, l'utilisation du *trace cache* ne donne strictement aucun gain, dans l'état actuel de nos technologies. Les programmeurs pourront en déduire une des innombrables raisons qui font que dérouler des boucles (*Loop unrolling*) est le mal incarné.

Le victim cache

Ce cache est un petit cache qui vient en supplément du cache principal (ou des caches L1, L2 et L3) et n'est présent que sur les caches de type *Write Back*. Il a été inventé pour limiter les défauts des caches *direct mapped*. Pour rappel, sur les caches *direct mapped*, chaque adresse mémoire est assignée à une seule et unique de cache : le contenu d'une adresse mémoire ne peut aller que dans cette ligne, et pas dans une autre.

Cela pose quelques problèmes : il arrive souvent que certaines adresses mémoires qui se partagent la même ligne de cache soient manipulée simultanément. Dans ce cas, tout accès à une de ces adresses va virer le contenu de l'autre adresse du cache. Nos deux adresses seront expulsées puis rechargées dans le cache continuellement : c'est ce qu'on appelle des *conflits miss*.

Pour limiter ces *conflits miss*, des scientifiques ont alors eu l'idée d'insérer un cache permettant de stocker les toutes dernières données virées du cache. En faisant ainsi, si une donnée est virée du cache, on peut alors la retrouver dans ce cache spécialisé. Ce cache s'appelle le ***Victim Cache***. Il va de soit que cette technique peut aussi être adaptée aux caches *N-Ways Associatives*.

Voyons comment est géré ce cache. Si jamais on veut accéder à une donnée qui n'est ni dans le cache, ni dans le *Victim Cache* : on charge notre donnée dans le cache, après avoir sauvegardée le contenu de celle-ci dans le *Victim Cache*. Ce *Victim Cache* est géré par un algorithme de suppression des lignes de cache de type FIFO : les données les plus anciennes du cache sont celles qui sont supprimée en cas de besoin. Si jamais on veut lire une donnée, et que celle-ci n'est pas dans le cache, mais qu'elle est dans le *Victim Cache* : on échange le contenu du *Victim Cache* avec la ligne de cache correspondante.

Petit détail : ce cache utilise un *Tag* légèrement plus long que celui du cache *Direct Mapped* au-dessus de lui. L'index de la ligne de cache doit en effet être contenu dans le *Tag* du *Victim Cache*, pour bien distinguer deux adresses différentes, qui iraient dans la même ligne du cache juste au-dessus.

Le TLB

Vous vous souvenez du *Translation Lookaside Buffer*, ou TLB, vu au chapitre précédent ? Et bien celui-ci est aussi un cache spécialisé, au même titre que le *Trace cache* ou le *Victim cache*. Ce cache est parfois découpé en plusieurs sous-caches, L1, L2, L3, etc. Sur les architectures Harvard, on trouve parfois deux TLB séparées : une pour les accès aux instructions, et une pour les accès aux données.

Le Prefetching

Dans le chapitre précédent, j'ai parlé du **principe de localité spatiale**, qui dit que nos programmes ont tendance à accéder à des données proches en mémoire. Pour profiter au maximum de ce principe de localité spatiale, il est plutôt avantageux de décider de précharger à l'avance les instructions ou données proches de celles chargées il y a peu : on permet au processeur de trouver plus souvent les données qu'il cherche dans le cache. Ce préchargement s'appelle le ***Prefetching***, et peut être effectué par le programmeur ou directement par le processeur. Le tout est de déterminer quelle donnée précharger, et quand !

Vous vous souvenez que les transferts entre la mémoire et le cache se font lignes de cache par lignes de cache. En conséquence, notre prefetcher va précharger des blocs qui ont la même taille que nos lignes de cache. Reste à savoir comment celui-ci se débrouille.

Certains processeurs mettent à disposition du programmeur une instruction spéciale capable de précharger des données à partir d'une certaine adresse. Cela permet aux programmeurs de gérer quelque peu le contenu du cache. Le problème, c'est qu'il faut programmer en assembleur pour pouvoir en profiter. Et soyons franc, les programmeurs actuels refusent de programmer en assembleur ou d'utiliser ces instructions dans leurs programmes, pour des raisons de portabilité.

Mais il existe une autre solution : laisser le processeur précharger lui-même les données. Pour ce faire, celui-ci contient souvent un circuit nommé ***Prefetcher***, qui s'occupe de précharger des données/instructions. Presque tous les processeurs grand public actuels possèdent un *Prefetcher*. Il faut dire que les gains de performances apporté par ce circuit sont assez impressionnants : se priver d'un tel avantage serait du suicide. Qui plus est, on peut utiliser à la fois des instructions de *Prefetch* et un *Prefetcher* matériel intégré au processeur : les deux solutions en sont pas incompatibles.

Mais comment ce *Prefetcher* peut-il fonctionner ?

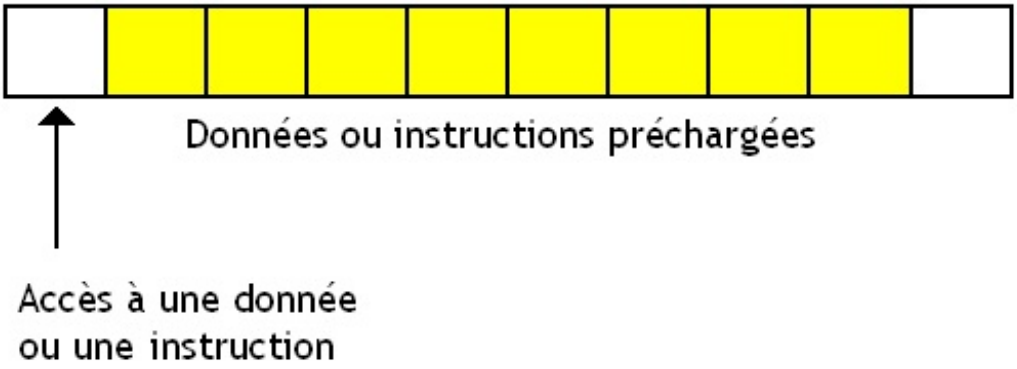
Tout d'abord, il faut préciser qu'il existe plusieurs types de *Prefetchers*. Certains sont assez rudimentaires, tandis qu d'autres sont plus évolués, plus efficaces. Ce chapitre va vous montrer ces diverses formes de *Prefetchers* et vous en expliquer les grands principes de fonctionnement.

Array Prefetching

Certains de ces *Prefetchers* sont particulièrement adaptés à l'utilisation de structures de données particulières : les **tableaux**. On a vu ce qu'était un tableau dans le chapitre sur l'assembleur, il y a quelques chapitres. Il s'agit de blocs de mémoires qu'on réserve pour y stocker des données de même taille et de même type. Certains de nos *Prefetchers* profitent du fait que ces tableaux sont souvent accédés cases par cases, avec certaines régularités. Je ne sais pas si vous avez déjà programmés, mais si c'est le cas, vous avez surement parcouru des tableaux cases par cases, en partant du début d'un tableau pour arriver à la fin. Et bien ce genre de parcourt est précisément accéléré par ces *Prefetchers*. Dès que l'on doit itérer dans un tableau, ces *Prefetchers* pourront servir. Voyons comment ceux-ci fonctionnent.

Prefetchers séquentiels

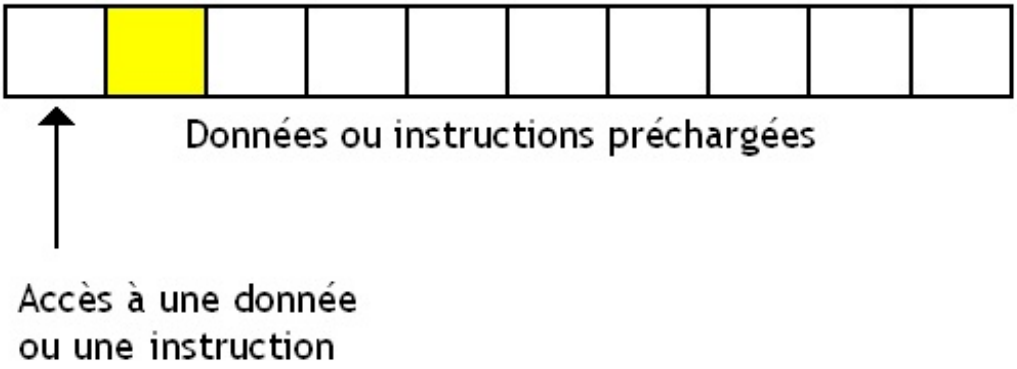
Commençons par les ***prefetchers* séquentiels**. Ces *prefetchers* se contentent de précharger les données immédiatement consécutives de la donnée venant tout juste d'être lue ou écrite.



Les *Prefetchers* de ce type sont vraiment stupides et ne fonctionnent que lorsqu'on accède à des données consécutives en mémoires, les unes après les autres. Mine de rien, ce genre d'accès à la mémoire est assez courant : cela arrive souvent quand on parcourir des tableaux dans leur totalité, en partant du début à la fin (ou de la fin vers le début).

One Block Lookahead

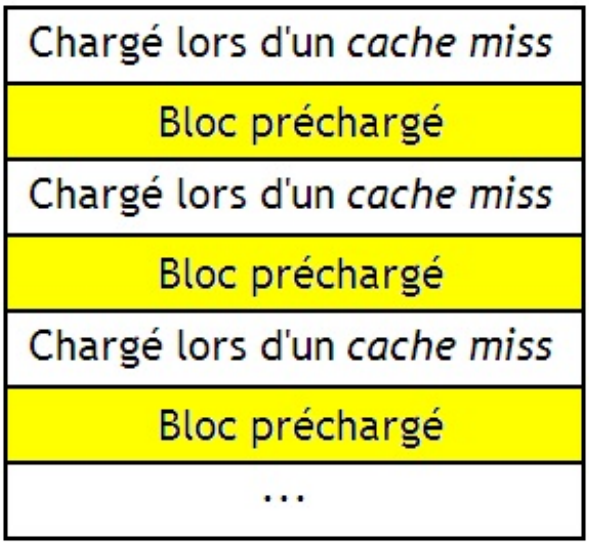
Le plus simple de ces *Prefetchers* sont ceux qui utilise le *Prefetching* de type ***One Block Lookahead***. Le terme est barbare, je me doute. Mais le principe est très simple. Il consiste simplement à précharger le bloc de mémoire qui suit. Ainsi, si vous accédez à un bloc de mémoire, le *Prefetcher* commencera à charger le bloc immédiatement suivant le plus tôt possible.



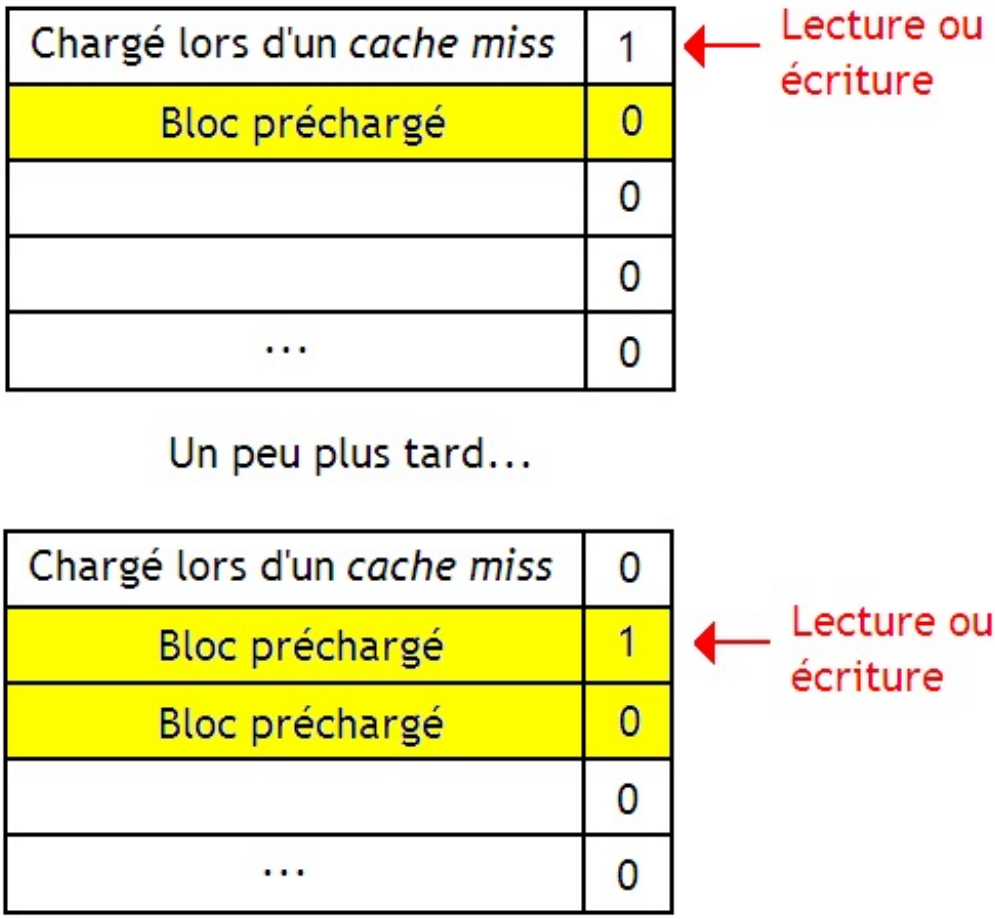
Pour fonctionner, ces *Prefetchers* ont juste besoin de se souvenir de la dernière adresse lue ou écrite, et de quoi déduire la prochaine adresse. Cette prédiction de la prochaine adresse se fait simplement en additionnant la longueur d'une ligne de cache à l'adresse du dernier bloc de mémoire lu ou écrit dans le cache. Le tout peut éventuellement être modifié de façon à ne pas prefetcher inutilement des blocs de mémoires déjà présents dans le cache.

Néanmoins, cette technique peut s'implémenter de trois façons différentes. Tout dépend de quand on *Prefetche*. La première solution consiste à Prefetcher le bloc suivant de manière systématique. Pour cela, rien de plus simple : il suffit simplement d'utiliser un compteur dans notre processeur. En plaçant notre adresse mémoire à lire ou écrire dans ce compteur, l'adresse suivante à prefetcher est calculée automatiquement.

Deuxième solution : ne prefetcher que lors d'un *cache miss*. Ainsi, si j'ai un *cache miss* qui me force à charger le bloc B dans le cache, le *Prefetcher* chargera le bloc immédiatement suivant avec.



Dernière solution : à chaque accès à un bloc de mémoire dans le cache, on charge le bloc de mémoire immédiatement suivant. Pour cela, on doit mémoriser quelle est la dernière ligne de cache qui a été accédée. Cela se fait en marquant chaque ligne de cache avec un bit spécial, qui indique si cette ligne a été accédée lors du dernier cycle d'horloge. Ce bit vaut 1 si c'est le cas, et vaut 0 sinon. Ce bit est automatiquement mit à zéro au bout d'un certain temps (typiquement au cycle d'horloge suivant). Le *Prefetcher* se contentera alors de charger le bloc qui suit la ligne de cache dont le bit vaut 1.



K-Bloc Lookahead Prefetching

Ce qui peut être fait avec un seul bloc de mémoire peut aussi l'être avec un nombre de bloc plus grand, fixé une bonne fois pour toute. Rien n'empêche de charger non pas un, mais deux ou trois blocs consécutifs dans notre mémoire cache. Ce qu'on a dit plus haut s'adapte alors. Mais attention : le nombre de blocs de mémoire chargés dans le cache est fixe : il ne varie pas. Le processeur va ainsi charger uniquement les 2, 3, etc; blocs consécutifs suivants. Et le nombre de blocs préchargés est toujours le même. Cela fonctionne bien si l'on utilise des données avec une bonne localité spatiale. Mais dans le cas contraire, cette technique fonctionne nettement moins bien et précharge beaucoup de blocs inutiles dans le cache. En comparaison, la technique qui ne charge qu'un seul bloc fonctionne nettement mieux dans ces cas là.

Adaptative Prefetching

Vous aurez aussi remarqué que ces *Prefetcher* ne fonctionnent que pour des accès à des zones consécutives de la mémoire. Le seul problème, c'est que beaucoup d'accès mémoire ne se font pas des zones de mémoire consécutives. L'utilisation d'un *Prefetcher* séquentiel est alors contre-productive. Pour limiter la casse, les *Prefetcher* sont capables de reconnaitre les accès séquentiels et les accès problématiques.

Cette détection peut se faire de deux façons. Avec la première, le *Prefetcher* va calculer une moyenne du nombre de blocs préchargés qui ont étés utiles. Ce calcul de cette moyenne se fera sur un échantillon contenant les N derniers blocs préchargés, histoire d'avoir une estimation locale. En clair : il va calculer le rapport entre le nombre de blocs qu'il a préchargé dans le cache, et le nombre de ces blocs qui ont étés accédés. Si jamais ce rapport diminue trop, cela signifie que l'on a pas affaire à des accès séquentiels : le *Prefetcher* arrêtera temporairement de précharger des trucs. Par contre, si jamais ce rapport dépasse une certaine limite, on est presque certain d'avoir affaire à des accès séquentiels : le *Prefetcher* préchargera des données.

Autre solution : garder un historique des derniers accès mémoires et de voir s'ils accèdent à des zones consécutives de la mémoire. Si c'est le cas, le *Prefetcher* prefetche, et dans le cas contraire, il ne prefetche pas. Le processeur peut même décider de désactiver temporairement le *Prefetching* si jamais le nombre de blocs préchargés utilement tombe trop prêt de zéro.

Streams Buffers

Le seul problème avec ces techniques de Prefetching, c'est le cas où le Prefetcher se trompe et où il prefetche des données qui n'auraient pas du l'être. Charger ces données va prendre du temps, nécessite d'accéder à la mémoire, etc. Mais surtout, des

données sont chargées inutilement dans le cache, et elles prennent la place d'une donnée qui aurait pu être plus utile. On va donc se retrouver avec quelques lignes de caches qui contiendront des données qui ne servent à rien, et qui auront éjecté des données potentiellement utiles du cache. De plus, cette donnée chargée inutilement ne va pas quitter le cache tout de suite. Si celui-ci utilise un algorithme de sélection des lignes de cache de type LRU, cette donnée chargée inutilement ne va pas être sélectionnée pour remplacement lors du prochain cache miss : ce seront d'autres lignes de caches qui le seront. Et il va de soit que ce lignes de caches auraient encore pu être utiles, contrairement à la donnée chargée à l'avance, qui squattera dans le cache.

Bref, on se retrouve avec une donnée inutile dans le cache, qui y reste durant longtemps, et qui se permet en plus de foutre dehors des données qui auraient pu se rendre utiles, elles ! C'est ce qu'on appelle un phénomène de ***Cache Pollution***. Il va de soit que limiter au maximum cette *Cache Pollution* est une nécessité si on veut tirer parti au maximum de notre mémoire cache. Reste à savoir comment.

Une première solution consiste à dire que les données prefetchées sont bonnes pour la casse si celles-ci ne servent pas rapidement. Vous savez que nos lignes de caches comportent des informations qui permettent de déterminer lesquelles ont été fréquemment utilisées (algorithme LFU), ou récemment utilisées (LRU). Et bien sur les processeurs modernes, une donnée prefétchée qui n'a pas servi peu de temps après son chargement anticipé depuis la mémoire est marquée comme étant très peu utilisée ou comme étant celle la moins récemment utilisée. Dans ce cas, cette donnée quittera le cache assez rapidement. Cela ne l'empêche pas d'écraser une donnée plus utile lors de son chargement anticipé, mais au moins, cela dure peu.

Autre solution : prefetcher nos données non pas dans le cache, mais dans une mémoire séparée, spécialisée dans le prefetching. Cette mémoire temporaire s'appelle un ***Stream Buffer***. En gros, tout ce qui est prefetché n'atterrit pas dans le cache, mais dans ce *Stream Buffer*. Si jamais un cache miss a lieu dans le cache, on regarde si la ligne de cache à lire ou écrire est dans le *Stream Buffer*. Si elle y est, on va la rapatrier dans le cache. Si ce n'est pas le cas, c'est que le *Stream Buffer* contient des données préfétchées à tord : le *Stream Buffer* est totalement vidé, et on va chercher la donnée en mémoire.

History based prefecther

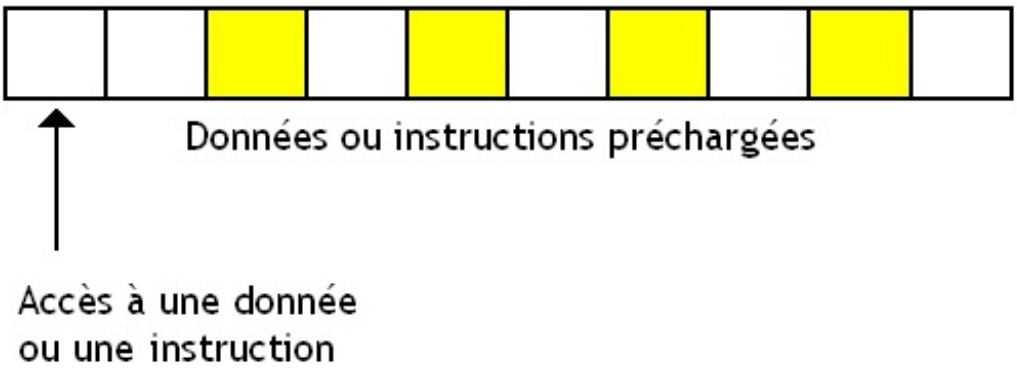
D'autres *prefetchers* sont un peu plus intelligents et sont capables de déduire quoi prefetcher en fonction des accès mémoires effectués juste avant. A partir de ces accès, ils peuvent voir s'il n'y a pas des régularités plus ou moins visibles dans ces accès qui pourraient les aider à déduire quelle sera la prochaine donnée accédée. Ces *prefetchers* doivent pour cela conserver un historique des accès mémoires effectués précédemment. Pour ce faire, ils utilisent une sorte de mémoire, qui stocke les dernières adresses mémoires accédées, qu'on appelle la ***Reference Prediction Table***.

Accès en Stride

Nos *Prefetchers* séquentiels. Les *Prefetchers* fonctionnent bien quand on parcourt des tableaux de données de base éléments par éléments. Mais il arrive que les programmeurs n'accèdent qu'à certains éléments du tableaux, tous séparés par une même distance. On obtient alors ce qu'on appelle des accès en *Stride*, qui ressemblent à ceci :

On le voit, ces accès se font sur des données séparées par une distance constante. On appelle de tels accès des accès en *K-Stride*, avec k la distance entre deux données accédées. Ces accès en *Stride* ont deux origines : les parcours de tableaux multidimensionnels, et le parcours de tableaux de structures/objets. Si on parcourt un tableau multidimensionnel de la mauvaise façon, (colonnes par colonnes en C, lignes par lignes en FORTRAN), on obtient de tels accès en *Stride*. D'un point de vue performances, ces accès sont souvent très lents. Il vaut mieux accéder à des données purement séquentielles. Moralité : programmeur, faites gaffe au sens de parcours de vos tableaux ! En plus, avec ce genre d'accès, un *Préfecther* séquentiel a tendance à charger des données inutiles (en blanc), situées entre les données utiles (celles en jaune). Cela fait pas mal de cache gaspillé, ce qui est synonyme de baisses de performances.

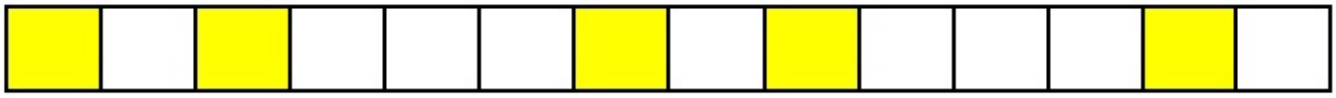
Pour éviter cela, on peut décider de transformer nos tableaux de structures en structures de tableaux ou modifier l'ordre de parcours de nos tableaux à deux dimensions. Mais pour limiter l'impact de tels accès, qu'ils soient dus à des programmeurs mal avertis ou non, certains *Prefetchers* gèrent de tels accès en *Stride* et peuvent prédire à la perfection quelle sera le prochain bloc de mémoire à charger. Cela ne rend pas ces accès aussi rapides que des accès à des blocs de mémoire consécutifs : on gâche souvent une ligne de cache pour n'en utiliser qu'une petite portion, ce qui n'arrive pas avec ces accès purement séquentiels. Mais cela aide tout de même beaucoup.



Néanmoins, il faut savoir que ces *prefetchers* ne peuvent fonctionner que si les accès à la mémoire sont assez rapprochés: si les données sont trop éloignées en mémoire, le *prefecther* ne peut rien faire et ne préchargera pas correctement les données. En dessous de cette limite, généralement, toutes les distances sont supportées. Le *prefetcher* peut ainsi choisir s'il faut précharger les données situées 250 adresses plus loin ou 1024 adresses plus loin, de façon à s'adapter le plus possible à la situation. Toutes les distances intermédiaire entre 0 et la limite maximale supportées par le *prefetcher* sont possibles.

Un peu mieux

Le gain apporté par les *prefetchers* capable de prédire des accès en *stride* est appréciable. Mais on pourrait certainement faire mieux. Par exemple, on peut vouloir faire en sorte que nos prefetchers fonctionnent à la perfection sur des accès cycliques ou répétitifs.



Des accès de ce type sont plus rares. Ils apparaissent surtout quand on parcourt plusieurs tableaux à la fois. Pour gérer aux mieux ces accès, on a inventé des *prefetchers* plus évolués, capables de ce genre de prouesses. Il en existe de plusieurs types, avec des performances différentes, mais on en parlera pas ici : le principe de fonctionnement de ces *prefetchers* est assez compliqué, et je ne suis pas sur que vous parler de chaines de Markov puisse vous intéresser. Mais malheureusement, ces *prefetchers* possèdent les défauts des *prefetchers* précédents : si on effectue des accès pas vraiment réguliers, notre prefetcher ne peut rien faire : il n'est pas devin.

Le Futur

Les techniques vues au-dessus sont celles qui sont couramment utilisées dans les processeurs actuels. Mais il en existe d'autres, qui n'ont pas encore étés utilisées couramment dans nos processeurs. Il existe ainsi des *prefetchers* plus évolués, capables de décider quelle donnée ou instruction charger en utilisant des méthodes plus compliquées. Ces *prefetchers* essayent de trouver des tendances, pas forcément répétitives, dans les accès mémoires précédents, et essayent de déduire quelle seront les adresses manipulées prochainement en effectuant des calculs statistiques plus ou moins évolués sur ceux-ci.

Linked Data Structures Prefetching

Les tableaux ne sont pas les seules structures de données utilisées par les programmeurs. Ces tableaux sont la structure de donnée idéale dans certaines situations, mais peuvent avoir des performances exécrables dans d'autres. Par exemple, il est très difficile d'ajouter de nouvelles valeurs dans un tableau ou d'en supprimer. La raison est simple : on ne sait pas ce qu'il y a avant ou après notre tableau en mémoire. Si on veut rajouter une donnée, peut importe qu'on veuille la placer avant ou après les autres : il y a un risque que cette donnée aille écraser une donnée préexistante. Rajouter ou supprimer des données ne peut se faire qu'en créant un nouveau tableau de la bonne taille, en recopiant les données du premier tableau dedans, et en supprimant l'ancien tableau.

Certaines applications ont besoin de structures de données qui permettent de supprimer ou d'ajouter un élément rapidement. Pour cela, on doit utiliser des alternatives aux tableaux. Suivant les besoins, la structure de donnée utilisée sera une liste chainée, un arbre, ou autre.

Linked Data Structures

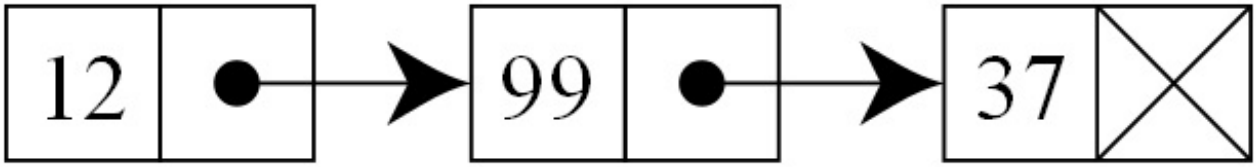
Dans tous les cas, ces structures de données sont toutes basées sur un principe simple : les données ne sont pas rassemblées dans un gros bloc de mémoire. A la place, elle sont crées séparément, et peuvent se retrouver à des endroits très éloignés en

mémoire.

Pour faire le lien entre les données, chacune d'entre elle sera stockée avec des informations permettant de retrouver la donnée suivante ou précédente en mémoire. Chaque donnée est donc stockée en mémoire, avec l'adresse de la ou des données suivantes ou précédentes.

Liste simplement chainée

Par exemple, on peut organiser nos données sous la forme d'une liste simplement chainée. Chaque donnée est ainsi stockée avec l'adresse de la donnée suivante. La fin de la liste est précisée par une adresse mémoire spéciale : c'est une adresse invalide, souvent nommée `null` ou `nil` dans les langages de programmation usuels.

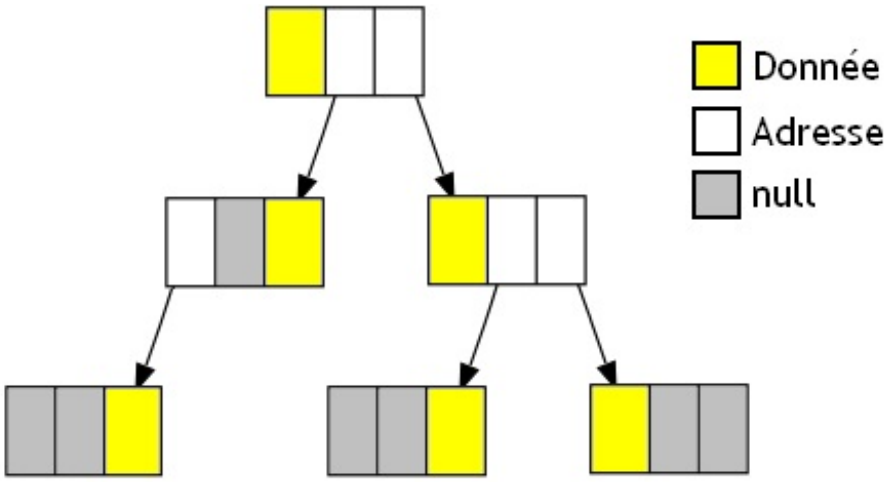


Liste doublement chainée

Pour faciliter le parcourt d'une telle liste, on peut aussi rajouter l'adresse de la donnée précédente. On obtient alors une liste doublement chainée.

Autres

Et on peut aller plus loin, en permettant à chaque donnée d'avoir plusieurs données suivantes et/ou précédentes. On obtient alors des arbres.



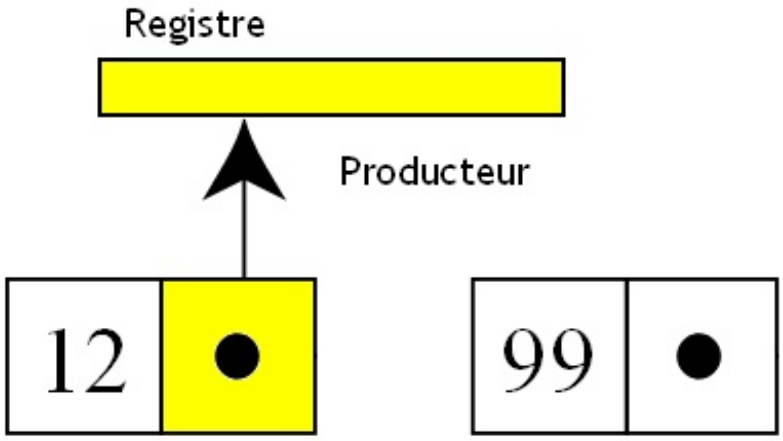
Et d'autres structures de données encore plus complexes existent : on pourrait citer les graphes, parler des différentes manières d'implémenter des arbres ou des listes, etc. Mais cela relèverait d'un cours d'algorithmique de haute volée.

Et le Prefetching ?

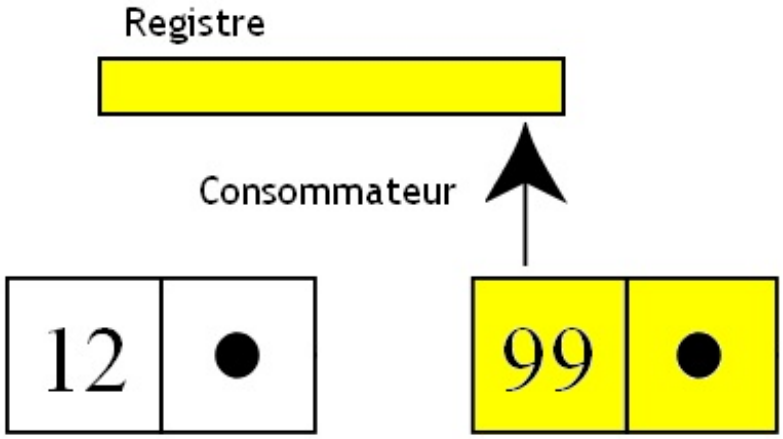
Il va de soit que le Prefetching séquentiel ou le Prefetching à base de Strides ne fonctionnent pas avec ces structures de données : nos données sont stockées n'importe comment en mémoire, et ne sont pas placées à intervalle régulier en mémoire. Cela dit, ça ne veut pas dire qu'il n'existe pas de techniques de Prefetching adaptée pour ce genre de structures de données. En fait, il existe deux grandes techniques pour ce genre de structures de données, et quelques autres techniques un peu plus confidentielles.

Dependance Based Prefetching

La première de ces techniques a reçu le nom de *Dependance Based Prefetching* . Cette technique se base sur un principe simple. Prenons un exemple : le parcourt d'une liste simplement chainée. Celui-ci s'effectue avec une boucle, qui parcourt les éléments les uns après les autres. Dans cette boucle, on trouve obligatoirement des instructions qui servent à passer d'une donnée à la suivante. Pour aller lire la donnée suivante, le processeur doit d'abord récupérer son adresse, qui est placé à coté de la donnée actuelle.



Puis, il doit charger tout ou partie de la donnée suivante dans un registre.



Comme vous le voyez, on se retrouve avec deux chargements de donnée depuis la mémoire : un premier qui produit l'adresse, et un autre qui l'utilise. Ces deux chargements de données sont effectuées par des instructions de lecture en mémoire. Dans ce qui va suivre, je vais identifier ces deux instructions en parlant d'instruction productrice (celle qui charge l'adresse), et consommatrice (celle qui utilise l'adresse chargée).

Table de corrélation

Avec le *Dependance Based Prefetching* , le processeur va tenter de prédire quels sont les chargements qui produisent ou consomment une adresse tel que vu au-dessus. Il va aussi détecter si une instruction va produire une adresse ou en consommer une. Pour cela, rien de plus simple : le processeur va regarder si deux instructions ont une dépendance producteur-consommateur : si c'est le cas, il s'en souviendra pour la prochaine fois. Pour s'en souvenir, il va utiliser une petite mémoire cache qui stockera les adresses du producteur et du consommateur. Cette petite mémoire cache s'appelle la **table de corrélation**.

Adresse du producteur	Adresse du consommateur
Adresse du producteur	Adresse du consommateur
Adresse du producteur	Adresse du consommateur
Adresse du producteur	Adresse du consommateur
Adresse du producteur	Adresse du consommateur
Adresse du producteur	Adresse du consommateur

Reste que les corrélations stockées dans cette table ne sortent pas de cuisse de Jupiter. Il faut bien que notre processeur puisse détecter les instructions productrices et consommatrices. Pour cela, notre processeur va les identifier lors de l'exécution d'une instruction consommatrice. Lorsque cette instruction consommatrice s'exécute, le processeur vérifie si l'adresse qu'elle cherche à lire a été chargée depuis la mémoire par une autre instruction. Si c'est le cas, alors le processeur détecte l'instruction consommatrice.

Potential Producer Windows

Avec cette technique, notre processeur doit se souvenir des dernières valeurs chargées depuis la mémoire, pour vérifier si elles servent d'adresse pour une instruction consommatrice. De plus, le processeur doit se souvenir de l'instruction qui a chargé cette donnée : sans cela, lors de la détection d'une instruction consommatrice, il ne peut pas savoir quelle est l'instruction productrice. Cela s'effectue avec une mémoire cache que l'on appelle la ***Potential Producer Windows***.

Donnée lue (adresse potentielle)	Adresse de l'instruction qui a chargé la donnée (producteur potentiel)
Donnée lue (adresse potentielle)	Adresse de l'instruction qui a chargé la donnée (producteur potentiel)
Donnée lue (adresse potentielle)	Adresse de l'instruction qui a chargé la donnée (producteur potentiel)

Fonctionnement

Le remplissage de la table de corrélation est alors très simple : lorsqu'une instruction de lecture se termine, on stocke la donnée qu'elle a lu depuis la mémoire, ainsi que l'adresse de l'instruction dans la *Potential Producer Windows*. Lors de l'exécution d'une prochaine instruction, on compare l'adresse à lire avec les données présentes dans la *Potential Producer Windows* : si il y a

correspondance avec un Tag, alors on détecte une correspondance producteur-consommateur. L'adresse de nos deux instructions productrice-consommatrice sont alors stockées dans la table de corrélation.

En quoi cela nous donne des indications pour pouvoir préfetcher quoi que ce soit ? Très simple : à chaque lecture, le processeur vérifie si cette lecture est effectuée par une instruction productrice en regardant le contenu de la table de corrélation. Dès qu'une instruction détectée comme productrice a chargé son adresse, le processeur se charge de prefetcher les données situées à cette adresse sans attendre que l'instruction qui consomme cette adresse s'exécute. Lorsqu'elle s'exécutera (quelques cycles plus tard), la donnée aura déjà commencé à être lue depuis la mémoire.

Runahead Data Prefetching

Enfin, il existe une dernière méthode purement matérielle pour précharger nos données. Il s'agit du préchargement anticipé. Cette technique est utile dans le cas où un processeur doit arrêter l'exécution de son programme parce que celui-ci attend une donnée en provenance de la mémoire. En clair, cette technique est utile si un *cache miss* à eu lieu et que le processeur n'est pas conçu pour pouvoir continuer ses calculs dans de telles conditions (pas de caches non-bloquants, pas d'exécution *Out Of Order*, etc).

Dans un cas pareil, le processeur est censé devoir stopper intégralement l'exécution de son programme. Mais à la place, on va continuer l'exécution des instructions suivantes de façon spéculative : on les exécute, même si on n'est pas censé avoir ce droit. Si elles accèdent à la mémoire, on laisse ces accès s'exécuter (sauf en écriture pour éviter de modifier des données alors qu'on n'aurait pas du): cela permettra d'effectuer des accès en avance et donc de précharger les données qu'elles manipulent. On continue ainsi tant que l'instruction qui a stoppé tout le processeur a enfin reçue sa donnée.

Le seul truc, c'est que tout doit se passer comme si ces instructions exécutées en avance n'avaient jamais eu lieu. Dans le cas contraire, on a peut-être exécuté des instructions qu'on aurait peut-être pas du, et cela peut avoir modifié des registres un peu trop tôt, ou mis à jour des bits du registres d'état qui n'auraient pas du être modifié ainsi. Il faut donc trouver un moyen de remettre le processeur tel qu'il était quand le cache miss a eu lieu. Pour cela, le processeur doit sauvegarder les registres du processeur avant d'exécuter spéculativement les instructions suivantes, et les restaurer une fois le tout terminé. Qui plus est, il vaut mieux éviter que ces instructions exécutées en avance puissent modifier l'état de la mémoire : imaginez qu'une instruction modifie une ligne de cache alors qu'elle n'aurait pas du le faire ! Pour cela, on interdit à ces instructions d'écrire dans la mémoire.

Les processeurs qui utilisent ce genre de technique sont redoutablement rares à l'heure où j'écris ce tutoriel. On trouve pourtant quelques articles de recherche sur le sujet, et quelques universitaires travaillent dessus. Mais aucun processeur ne *prefetche* ses données ainsi. Il y a bien le processeur Rock de la compagnie Sun, qui aurait pu faire l'affaire, mais celui-ci a étè annulé au dernier moment.

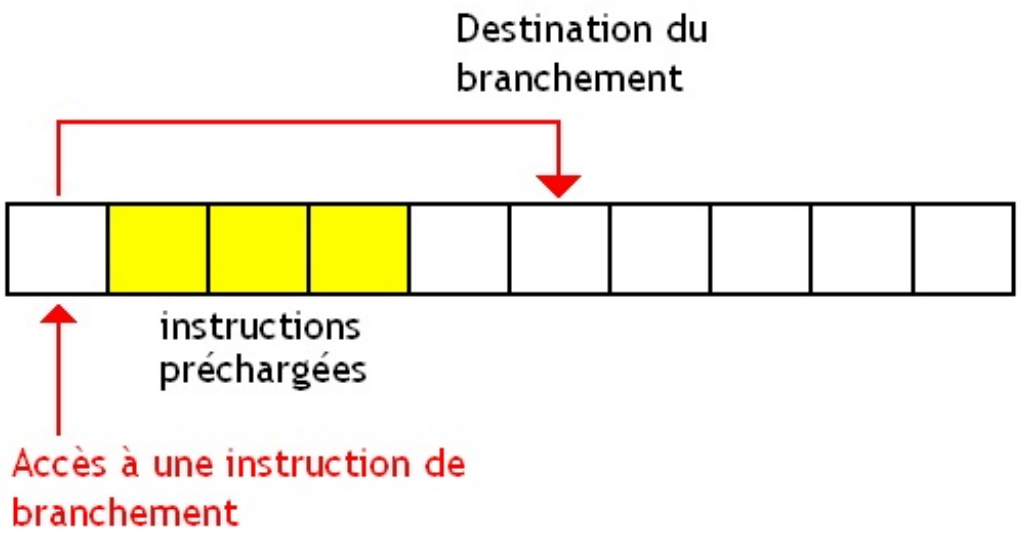
Instruction Prefetching

Au fait : sur certains processeurs, on utilise deux *prefetchers* séparés : un pour les instructions, et un autre pour les données. Cela permet d'adapter plus facilement les stratégie de préchargement. En effet, les instructions ont parfois besoin de techniques de préchargement spécialisées qu'on va voir dans ce qui suit. Les *prefetchers* spécialisés pour les instructions peuvent donc se contenter de techniques de *préfetching* adaptées, assez simples, tandis que ceux spécialisés pour les données peuvent utiliser des techniques plus compliquées. Cela évite au *prefetcher* de confondre accès aux instructions et accès aux données : si jamais le *prefetcher* confond un accès à une instruction et utilise une technique pas adaptée, cela pourrait lui faire rater un préchargement.

Prefetching séquentiel, le retour !

Le *prefetching* séquentiel est parfaitement adapté au préchargement des instructions d'un programme, vu que ses instructions sont placées les unes après les autres en mémoire.

Néanmoins, il arrive que dans certains cas, le processeur doivent sauter à un endroit différent du programme, et ne passe pas directement à l'instruction consécutive en mémoire : lorsque l'on exécute un branchement. Avec un *prefetcher* purement séquentiel, on peut se retrouver avec quelques problèmes lorsque l'on exécute un branchement : celui-ci aura tendance à précharger les instructions située après le branchement, qui ne sont pas censée être utilisées si le branchement renvoie le processeur ailleurs.

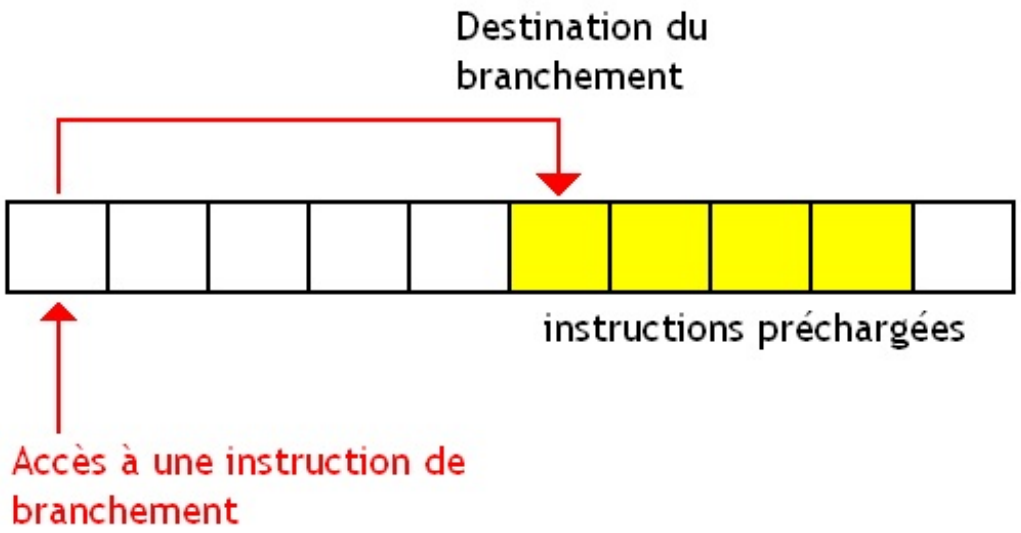


Les branchements qui posent ce genre de problèmes sont : tous les branchements inconditionnels, et notamment les appels et retour de sous-programmes, ainsi que les branchements conditionnels pris (qui envoient le processeur ailleurs, et non à la suite du programme). Autant le dire tout de suite : ils sont assez nombreux.

Target Line Prefetching

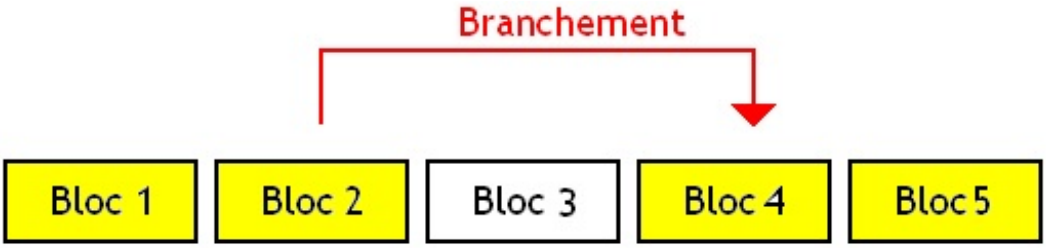
Il serait judicieux de trouver des moyens qui permettent de limiter la casse. Pour ce faire, il faudrait trouver un moyen de savoir où va nous faire atterrir notre branchement. Ainsi, notre prefetcher aurait un moyen de savoir s'il faut charger les instructions placées immédiatement après le branchement ou les instructions situées ailleurs. Si on ne connait pas l'adresse de destination du branchement, le *prefetcher* ne sait pas quoi précharger, et doit se rabattre sur du *prefetching* sequentiel.

Néanmoins, il peut déduire cette adresse dans le cas de certains branchements inconditionnels : les adresses des appels de sous-programmes, retour de sous-programmes, branchements inconditionnels directs (dont l'adresse est fixe), peuvent être prédites à la perfection. Cette adresse est fixe, et ne varie jamais. Une fois qu'on connait celle-ci, il suffit de la mémoriser et de s'en souvenir pour la prochaine fois. C'est ce qu'on appelle le ***Target Line Prefetching***.



Pour implémenter cette technique, nos *Prefetchers* incorporent une sorte de petite mémoire cache, capable de stocker ces adresses de destination pour ces branchements. Plus précisément, cette mémoire cache va contenir des correspondance entre : une ligne de cache, et la ligne de cache à charger à la suite de celle-ci.

Pour donner un exemple, regardons ce qui se passe avec le morceau de programme suivant.



Dans cet exemple, la mémoire cache va avoir le contenu suivant :

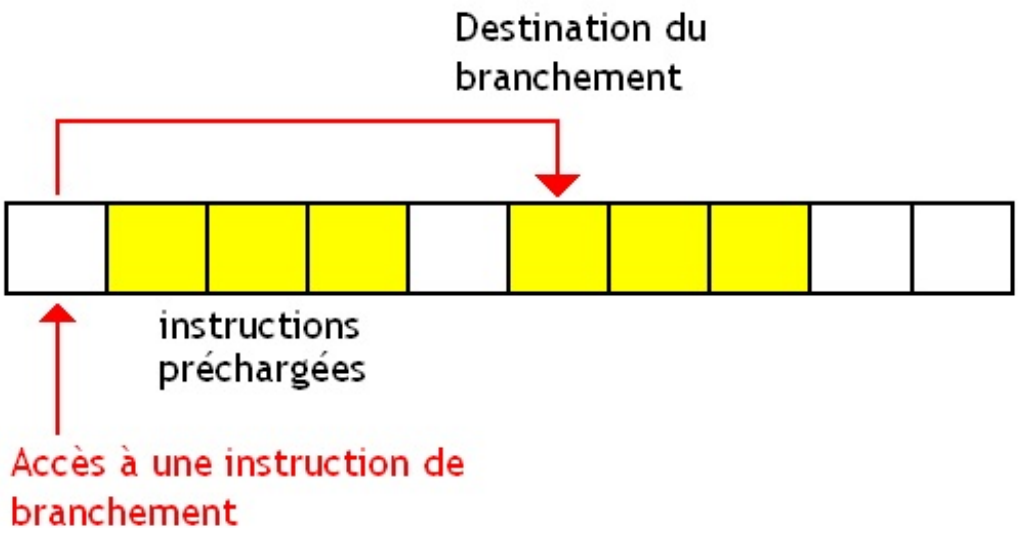
1	2
2	4
4	5

Cette technique de Prefetching fonctionne sur un principe très simple : quand on lit ou écrit dans une ligne de cache, le processeur interroge la table de correspondance. Il récupère la ligne de cache suivante, et préfetche celle-ci.

Pour plus d'efficacité, certains processeurs utilisent cette table d'une manière un peu plus efficace. Dans ceux-ci, la table ne stocke pas les correspondance entre lignes de cache si celles-ci sont consécutives. Si jamais deux lignes de caches sont consécutives, on fera alors face à un Cache Miss dans cette mémoire qui stocke nos correspondances. Le Prefetcher utilisera alors automatiquement un Prefetching séquentiel. Ainsi, la table de correspondance est remplie uniquement avec des correspondances utiles.

Wrong Path Prediction

On peut améliorer la technique vue juste avant pour lui permettre de s'adapter au mieux aux branchements conditionnels. En plus de charger les instructions correspondant à un branchement pris, qui envoie à l'autre bout du programme, on peut aussi charger en même temps les instructions situées juste après les branchement. Comme ça, si le branchement en question est un branchement conditionnel, les instructions suivantes seront disponibles, que le branchement soit pris ou pas. On appelle cette technique la ***Wrong Path Prefetching***.



Il va de soit que l'adresse de destination du branchement n'est connue que lors de l'étage de Decodage de l'instruction : il faut théoriquement que le branchement aie fourni son adresse avant que l'on puisse l'utiliser. Ce qui fait qu'au final, l'adresse du bloc à prefetcher dans le cache est connue très tard. Si le branchement est pris, il n'y a pas besoin de prefetcher : notre processeur fetchera normalement les instructions de destination indiquées par le branchement. Mais si le branchement n'est pas pris

(condition fausse), on prefetchera quand même les instructions correspondant à un branchement non-pris (celles situées juste après les branchement).

A première vue, ce chargement est inutile : on prefetche des instructions qui ont étés zappées par le branchement. L'astuce vient du fait que notre branchement peut parfaitement s'exécuter plusieurs fois : un programme est souvent rempli de boucles, fabriquées avec des branchements, sans compter que ces boucles peuvent aussi contenir des branchements. Et l'adresse de destination du branchement peut changer lors de son exécution. Avec la *Wrong Path Prefetching* , on va charger les instructions correspondant aux deux cas : branchement pris (condition vraie), ou non-pris (condition fausse). Ainsi, lors de la seconde exécution du branchement, les instructions de destination seront forcément dans le cache.

D'après certaines simulations, cette technique fonctionne un peu mieux que le *Target Line Prefetching* dans certaines situations. Il suffit d'avoir beaucoup de branchements conditionnels répétés dans notre programme, et une mémoire RAM avec un bon débit pour obtenir de bons résultats.

Autres

Il existe encore d'autres techniques de *Prefetching* pour nos instructions. On peut notamment citer les techniques basées sur des chaines de Markov, qui utilisent des statistiques pour déduire si un futur branchement sera pris ou pas pour décider quelle bloc de mémoire charger dans le cache. Mais on n'en parlera pas ici. Du moins, pas encore !

Partie 7 : Le parallélisme d'instruction et les processeurs modernes

Pour commencer notre exploration des diverses améliorations inventées par les fabricants de composants informatiques, on va commencer par notre processeur. Et on va voir que c'est ce processeur qui a subit le plus d'améliorations au fil des ans. A chaque nouveau processeur mit en vente par Intel ou AMD, il y a toujours une amélioration de faite, plus ou moins importante. Dans ce chapitre, on ne va pas tout passer en revue, mais on voir les améliorations les plus importantes. Vous allez voir que la majorité des améliorations du processeur tournent autour de deux concepts : exécuter un maximum d'instructions en parallèle (simultanément).

Le pipeline : qu'est-ce que c'est ?

Concevoir un processeur n'est pas une chose facile. Et concevoir un processeur rapide l'est encore moins, surtout de nos jours. Dans les chapitres précédents, on a évoqué le fonctionnement d'un ordinateur, mais on n'a pas encore parlé de ses performances, de sa rapidité. Nous allons malheureusement devoir enfin parler de performances et mettre les mains dans le cambouis pour comprendre comment fonctionnent les processeurs modernes. En effet, toutes les optimisations des architectures actuelles répondent toutes au même besoin : créer des processeurs plus rapides. Pour commencer, nous allons devoir déterminer ce qui fait qu'un programme lancé sur notre processeur va prendre plus ou moins de temps pour s'exécuter.

Un besoin : le parallèlisme d'instruction

Le temps que met un programme pour s'exécuter dépend de pas mal de choses. Il dépend du temps mit au processeur pour exécuter ses instructions, du temps passé à attendre des données en provenance de cette feignasse de mémoire, et du temps passé à attendre que ces lourdeaux de périphériques fassent ce qu'on leur demande ! Dans la réalité, vous pouvez être certains que votre programme passera les 3/4 de son temps à attendre la mémoire et les périphériques : ceux-ci sont hyper-lents comparés à la mémoire, à tel point qu'il est impossible de créer des programmes rapides sans prendre en compte la hiérarchie mémoire.

Mais dans ce chapitre, on ne va s'intéresser qu'au temps nécessaire pour exécuter les instructions d'un programme, en considérant que le temps mit pour accéder à la mémoire ou aux périphériques ne comptent pas, afin de se simplifier la vie ! Mais rassurez-vous : les temps d'attentes dus à la mémoire et aux périphériques auront un chapitre rien que pour eux.

Le temps pris par notre programme pour exécuter ses instruction, qu'on notera Ti, dépend :

- du nombre moyen N d'instructions exécutées par notre programme (ce nombre peut varier suivant les données manipulées par le programme, leur taille, leur quantité, etc) : celui-ci porte un petit nom : il s'appelle l'*Instruction path length*) ;
- du nombre moyen de cycles d'horloge nécessaires pour exécuter une instruction, qu'on notera CPI (ce qui est l'abréviation de *Cycle Per Instruction*) ;
- et de la durée P d'un cycle d'horloge.

Ce temps T est égal à :

$$Ti = N \times CPI \times P$$

Quand on sait que la fréquence n'est rien d'autre que l'inverse de la période d'un cycle d'horloge, on peut récrire cette équation comme ceci :

$$Ti = \frac{N \times CPI}{Frequence}$$

Comme on le voit, il existait trois solutions pour rendre un programme plus rapide :

Que faire ?	Comment ?
Diminuer son nombre d'instructions N	Pour cela, il existe plusieurs solutions : • **compter sur le programmeur** pour optimiser son programme : un bon choix d'algorithme et une implémentation efficiente de celui-ci peut donner des gains assez intéressants ; • **améliorer le jeu d'instruction**. Cela peut se faire en créant des instructions plus complexes, capables de remplacer des suites d'instructions simples : il n'est pas rare qu'une grosse instruction complexe fasse exactement la même chose qu'une suite d'instructions plus élémentaires. C'est la raison même de l'existence des

	processeurs CISC.
Diminuer CPI	Il existe différentes solutions pour diminuer CPI. Tout d'abord, on peut concevoir notre processeur de façon à **diminuer le temps mis par notre processeur pour exécuter une instruction**. C'est particulièrement difficile et nécessite de refaire les circuits de notre processeur, trouver de nouveaux algorithmes matériels pour effectuer une instruction, améliorer le fonctionnement de notre processeur et sa conception, etc. De ce point de vue, les processeurs RISC sont avantagés : leurs instructions sont toutes simples, et peuvent souvent s'effectuer en quelques cycles d'horloges. On est vraiment loin des anciens processeurs CISC : si certaines instructions de base étaient simples et rapides, d'autres instructions très complexes pouvaient mettre une bonne dizaine voire centaine de cycles d'horloge pour s'exécuter. Un autre solution consiste à **mieux choisir les instructions** utilisées. Comme je l'ai dit plus haut, le nombre CPI est un nombre moyen : certaines instructions sont plus rapides que d'autres. En utilisant de préférence des instructions rapides au lieu d'instructions plus lentes pour faire la même chose, on peut facilement diminuer le CPI. De nos jours, les programmeurs n'ont que très peu d'influence sur le choix des instructions à utiliser : les langages de haut niveau comme le C++ ou le Java se sont démocratisés et ont délégués cette tache aux compilateurs (qui se débrouillent particulièrement bien, en passant).
Augmenter la fréquence	Pour cela, il faut utiliser des composants électroniques plus rapides. Généralement, cela nécessite de miniaturiser les transistors de notre processeur : plus un transistor est petit, plus il est rapide (rappelez vous le chapitre sur les transistors). Durant un temps, cela fonctionnait à merveille, grâce à la loi de Moore. Mais de nos jours, le ***Heat wall*** et le ***Memory wall*** ont finis par diminuer les effets de la montée en fréquence, rendant celle-ci de plus en plus inutile en terme de performances.

Mais avec le temps, il est devenu de plus en plus difficile de monter en fréquence. Le coup de grâce fut celui porté par le ***Heat wall*** et le ***Memory wall*** : la montée en fréquence devenant de plus en plus difficile, malgré les améliorations des processus de fabrication des composants électroniques, et les performances ont commencées à stagner.

La solution consistant à rajouter des instructions et modes d'adressage complexes ne fonctionnait pas non plus : ces instructions étaient souvent assez complexes et les occasions de les utiliser étaient assez rares. Pourtant les performances pouvaient être au rendez-vous, mais utiliser ces instructions est souvent difficile pour un compilateur. Mais le principal problème de cette solution n'est pas là : modifier un jeu d'instruction est quelque chose qui doit se faire sur le long terme, avec le poids de la compatibilité, ce qui n'est pas sans poser quelques problèmes.

Par exemple, un programme qui utiliserait des instructions toutes nouvelles ne peut pas fonctionner sur les anciens processeurs ne possédant pas ces instructions. Ainsi, on a beau rajouter des tas d'instructions dans nos processeurs, il faut attendre assez longtemps pour que celle-ci soient utilisées : le temps que presque tout le monde aie un processeur capable d'exécuter ces nouvelles instructions. Pour donner un exemple très simple : à votre avis, sur les nouvelles instructions rajoutées dans chaque nouveau processeur Intel, AMD, ou ARM, combien sont réellement utilisées dans les programmes que vous utilisez ? Combien utilisent les instructions SSE, AVX ou autres extensions récentes ? Surement peu.

Il ne restait plus qu'une solution : diminuer le CPI. Cela n'était pas facile, et diverses solutions ont étés mises en place : améliorer les circuits de calcul et le séquenceur de notre processeur, fabriquer des circuits plus rapides, etc. Quoiqu'il en soit, les concepteurs de processeurs ont cherché à optimiser au mieux les instructions les plus utilisées et se sont plus ou moins heurtés à un mur : leurs opérations étaient déjà diablement rapides au point qu'il était assez difficile de leur faire prendre moins de temps.

En clair, les solutions orthodoxes visant à toucher au jeu d'instruction, à la fréquence ou aux circuits de l'unité de calcul ont atteint une sorte de mur difficile à dépasser. Les concepteurs de processeurs ont donc dût ruser et ont du trouver d'autres méthodes. Il est devenu évident au fil du temps qu'il fallait réfléchir hors du cadre et trouver des solutions innovantes, ne ressemblant à rien de connu. Ils ont fini par trouver une solution assez incroyable : **exécuter plusieurs instructions en même temps** ! Comme ça, pas besoin de gaspiller son énergie à rendre nos instructions encore plus rapides en améliorant des circuits proches de l'optimum : avec cette méthode, le CPI devenait inférieur à 1 pour les instructions rapides, ce qui donnait de gros gains en performances.

Pour exécuter plusieurs instructions en même temps, il a bien fallu trouver quelques solutions diverses et variées. Le pipeline est une de ces solutions. Pour expliquer en quoi il consiste, il va falloir faire un petit rappel sur les différentes étape d'une instruction.

Le pipeline : rien à voir avec un quelconque tuyau à pétrole !

Ceux qui se souviennent du chapitre sur la micro-architecture d'un processeur savent qu'une instruction est exécutée en plusieurs étapes bien distinctes. Suivant son mode d'adressage, ou les manipulations qu'elle doit effectuer, notre instruction va devoir passer à travers plusieurs étapes. Généralement, on trouve 7 grandes étapes bien distinctes :

- il faut charger d'instruction depuis la mémoire : c'est l'étape de ***Fetch*** ;
- puis il faut la décoder : c'est l'étape de ***Decode*** ;
- puis, certains modes d'adressages demandent de calculer l'adresse à laquelle aller lire nos opérandes ;
- si besoin, nos opérandes sont lues depuis la mémoire : c'est l'étape d'***Operand Fetching*** ;
- notre instruction effectue un calcul ou un échange de donnée entre registres : c'est l'étape d'***Exec*** ;
- puis, certains modes d'adressages demandent de calculer l'adresse à laquelle aller stocker notre résultat ;
- et enfin, le résultat est écrit en mémoire : c'est l'étape de ***Writeback***.

Ces étapes sont parfois elles même découpées en plusieurs sous-étapes. De même, l'étape de décodage peut être scindée en deux sur les processeurs utilisant des instructions de taille variable : on rajoute une étape pour déterminer la taille de notre instruction avant de la décoder.

De même, rien n'empêche de regrouper certaines de ces sous-étapes : on peut par exemple, regrouper l'étape d'*Operand Fetching* avec celle qui est chargée de calculer l'adresse de la donnée à charger. Par exemple, rien n'empêche d'utiliser un processeur qui utilise 5 étapes :

- **Fetching** : on charge l'instruction depuis la mémoire ;
- **Décodage** : on décode l'instruction ;
- **Exécution** : on effectue un calcul ;
- **Operand Fetching** : on effectue une lecture en mémoire ;
- **Write back** : on écrit le résultat de l'instruction dans un registre ou en mémoire.

Et enfin, rien ne nous empêche de rajouter des étapes supplémentaires : on le fera dans la suite de ce tutoriel. Vous verrez qu'on rajoutera des étages nommés Issue, Dispatch, Rename, et Commit pour ne citer que ceux-là.

Sans pipeline

Quoiqu'il en soit, ces étapes sont plus ou moins indépendantes, mais sont exécutées l'une après l'autre, dans l'ordre, sur un processeur sans pipeline.

Sans pipeline, on est obligé d'exécuter les instructions les unes après le autres. Ce n'est qu'une fois une instruction terminée qu'on peut passer à la suivante. Dit autrement, on ne peut commencer à exécuter une instruction que lorsque la dernière étape de l'instruction précédente est terminée.

Et dieu inventa le pipeline

En fait non, c'est pas lui ! L'inventeur du pipeline s'appelle David Patterson : ceux qui ont une bonne mémoire se rappelleront que cet homme est l'inventeur des architectures RISC, qui furent parmi les premiers processeurs à incorporer un pipeline.

Le pipeline répond à un but précis : avec un processeur sans pipeline, on doit attendre qu'une instruction soit finie pour exécuter la suivante. Avec un pipeline, on peut commencer à exécuter une nouvelle instruction sans attendre que la précédente soit terminée.

Pour voir comment c'est possible, il faut remarquer que les étapes de nos instructions sont totalement indépendantes. Ainsi, la première étape d'une instruction peut commencer pendant que l'instruction précédent passe à la suivante. Par exemple, on pourrait *fetcher* la prochaine instruction pendant que l'instruction en cours d'exécution en est à l'étape d'*Exec*. Après tout, ces deux étapes sont complètement indépendantes et utilisent des circuits séparés. En regardant bien, on remarque que chaque étape d'une instruction est indépendante des étapes précédentes. Il est donc possible de faire en sorte que chaque étape puisse commencer à traiter une nouvelle instruction pendant que la précédente passe à l'étape suivante. C'est le principe du pipeline : **exécuter plusieurs instructions différentes, chacune étant à une étape différente des autres**.

Avec un pipeline, chaque instruction est découpée en plusieurs étapes, chacune étant effectuée par une (ou plusieurs) unité de calcul séparées. Cela permet d'avoir des étapes réellement indépendantes : si deux instruction dans des étapes différentes ont besoin du même circuit, on ne peut pas exécuter les deux dans des étapes différentes en même temps, et une d'entre elles doit être choisie. Créer des circuits spécialisés pour chaque étape est donc un premier pas. Chacun des circuits permettant d'effectuer une des étapes d'une instruction est alors appelé un **étage du pipeline**.

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Dans l'exemple du dessus, on a un pipeline à 5 étages. Comme vous le voyez, durant le cycle noté 4, 4 instructions différentes s'exécutent : la première est à l'étape MEM, la seconde à l'étape EX, la troisième à l'étape ID, la quatrième à l'étape IF, et un étage ne sert à rien (il aura pu servir et on aurait alors eu 5 instructions exécutées en même temps).

Le nombre total d'étapes nécessaires pour effectuer une instruction (et donc le nombre d'étages du pipeline) est appelé la **profondeur du pipeline**. Plus ce nombre est grand, plus notre pipeline peut exécuter d'instructions en même temps. Du moins en théorie, parce qu'on va voir qu'il y a quelques petites subtilités qui viennent mettre leur grain de sel. Par exemple, dans l'exemple du dessus, un étage était inutilisé : et bien sachez que pour diverses raisons qu'on abordera plus tard, ce genre de cas est possible, et est même fréquent.

Pour les curieux, voici les longueurs de pipeline de certains processeurs plus ou moins connus.

Processeur	Longueur du pipeline
Motorola PowerPC G4	7
MIPS R4400	8
Intel Itanium	10
Intel Pentium II	14
Intel Pentium 3	10
Intel Pentium 4 Prescott	31
Intel Pentium 4	20
AMD Athlon	12
AMD Athlon 64	16
IBM PowerPC 970	16
Sun UltraSPARC IV	14

Etages, circuits et fréquence

Concevoir un processeur incorporant un pipeline ne se fait pas simplement et nécessite quelques modifications de l'architecture de notre processeur.

Un besoin : isoler les étages du pipeline

Tout d'abord, chaque étape d'une instruction doit être exécutée indépendamment des autres. Pour cela, nos étages, chargés

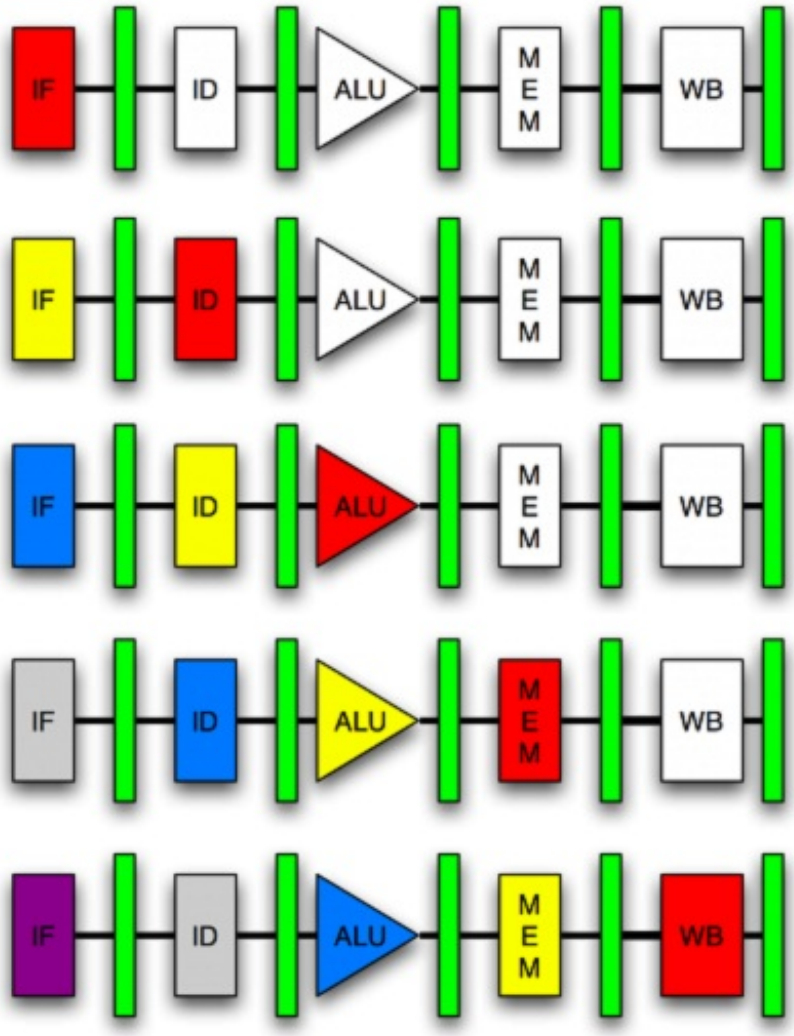
chacun de l'exécution d'une étape, doivent utiliser des circuits indépendants : impossible de réutiliser un circuit dans plusieurs étapes.

Dans un processeur sans pipeline, plusieurs étapes différentes d'une même instruction peuvent utiliser le même circuit. Par exemple, le circuit chargé d'effectuer l'addition (l'additionneur) peut être utilisé :

- pour augmenter la valeur du registre d'adresse d'instruction durant l'étape de *fetch*, afin de le faire pointer sur l'adresse de l'instruction suivante ;
- pour calculer l'adresse d'une opérande si besoin est (généralement, les calculs d'adresses sont des basées sur des additions, des multiplications et éventuellement des décalages) ;
- puis pour effectuer l'addition lors de l'étape d'*exec*;
- pour calculer l'adresse à laquelle sauvegarder le résultat de l'instruction d'addition si besoin.

Cela ne pose pas le moindre problème : après tout, ces étapes ne sont jamais simultanées sur de tels processeurs et partager le même circuit pour des étapes différentes permet d'éviter de dupliquer des circuits. Dans notre exemple avec l'additionneur, cela permet d'utiliser un seul additionneur au lieu de 3 ou 4.

Mais sur un processeur doté de pipeline, on ne peut se permettre ce genre de chose. Il est préférable que chaque étape aie son propre unité de traitement dédiée pour éviter à différentes étapes de se partager le circuit. Imaginez que deux instructions dans des étapes différentes aient besoin du même circuit : il est impossible de partager le circuit en deux et d'en donner la moitié à chaque instruction. Un processeur utilisant un pipeline utilisera donc beaucoup plus de transistors et de portes logiques pour fonctionner, ce qui a un certain cout.



Mine de rien, cette séparation ne fait pas tout : certaines ressources, comme les registres ou le bus mémoire peuvent être partagées entre instructions : sur un processeur doté de pipeline, deux instructions peuvent vouloir manipuler le même registre ou le bus mémoire, et il faudra gérer ce genre de cas avec des techniques plus ou moins élaborées.

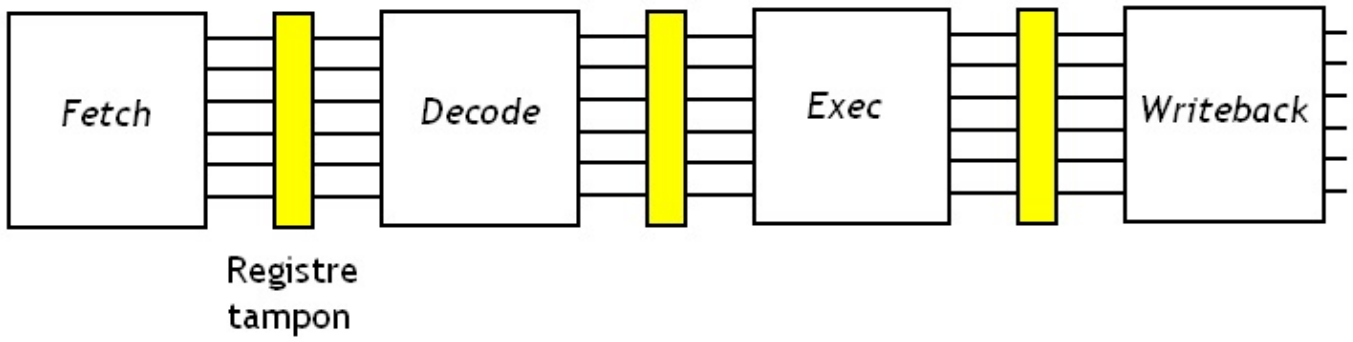
On comprend qu'il est important de séparer les unités en charge de chaque étape. Il existe pour cela diverses approches qu'on va

vous détailler dans ce qui suit.

Comment on fait ?

Certains pipelines intercalent des registres entre chaque étage du pipeline pour les rendre indépendants. Ces registres jouent le rôle de tampon et isolent chaque étage des autres.

Exemple avec un pipeline à 4 étages.

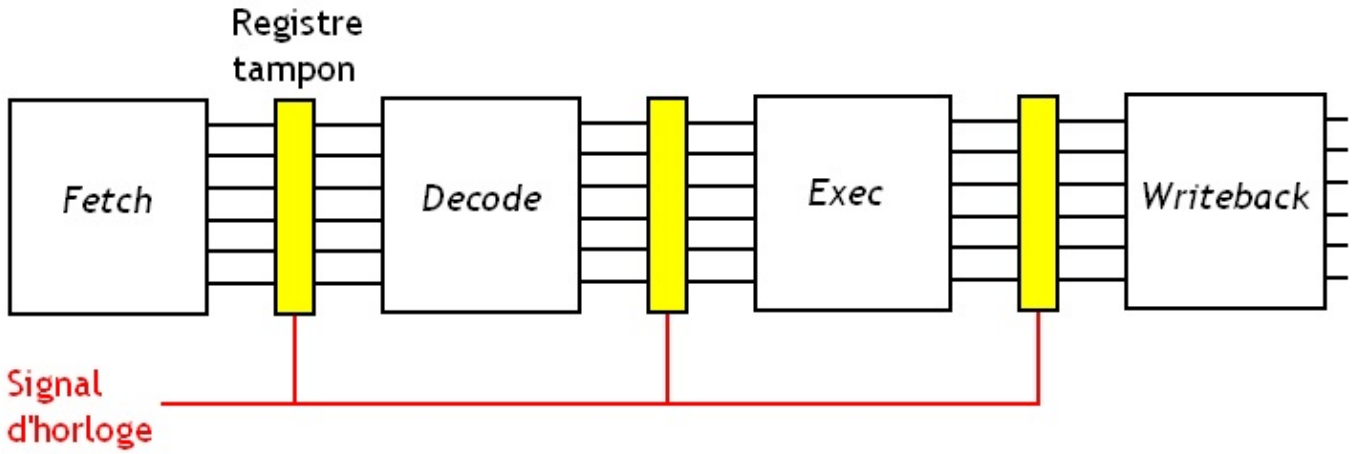


Ces registres sont des registres complètements invisible pour le programmeur : on ne peut pas manipuler ceux-ci avec des instructions du type `load`, `store`, ou d'autres instructions censée pouvoir manipuler le contenu d'un registre. Il n'ont donc pas de nom, et sont accessibles que par les unités en charge de chaque étape.

Quand une unité à fini son travail, elle écrit son résultat dans le registre. Quelques instants plus tard, l'unité en charge de l'étape suivante lira le contenu de ce registre et pourra alors effectuer l'étape qui lui est attribuée. Le tout est de savoir quand l'unité suivante lira le contenu de ce registre.

Buffered Synchrones

Sur certains processeurs, le pipeline est synchronisé sur l'horloge de notre processeur. Chaque étage du pipeline met donc un cycle d'horloge pour effectuer son travail. Il va lire le contenu du registre qui le relie à l'étape qui le précède au début du cycle et va déduire le résultat qu'il écrira dans le registre qui le relie à l'unité suivante juste avant le prochain cycle d'horloge.



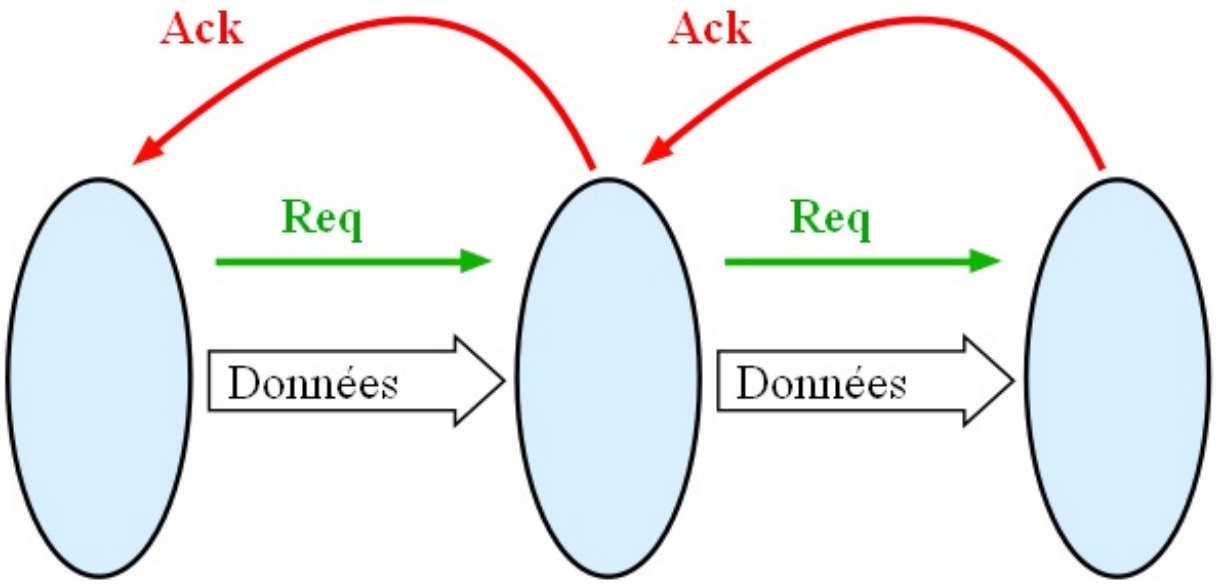
Pour éviter les ennuis, on fait en sorte que la fréquence de l'horloge soit suffisamment petite pour éviter que le contenu d'un registre n'aie pas fini d'être lu (par l'étage le suivant) avant d'être modifié par l'étage qui le précède.

Ce sont ce genre de pipeline que l'on trouve dans les processeurs Intel et AMD les plus récents.

Buffered Asynchrones

Sur d'autres pipelines, il n'y a pas d'horloge pour synchroniser le passage des résultats d'un étage à un autre. Sur ces

processeurs, chaque étage est relié par deux fils à l'étage suivant : on nommera ces fils REQUEST et ACK.



Quand un étage a fini son travail, il envoie un 1 à l'étage suivant sur le fil REQ, pour lui dire : " j'ai fini mon travail, et j'ai quelques données pour toi " . Il va alors attendre que l'étage suivant soit libre et que celui-ci lui réponde : " je suis libre, je m'occupe de tes données " en mettant le fil ACK à 1. Suite à cela, ces deux fils sont remis à zéro et nos étages peuvent se remettre au travail chacun de leur coté.

Unbuferred pipeline

D'autres pipelines se débrouillent sans registres intercalés entre chaque étage, mais ceux-ci sont beaucoup plus rares.

Une histoire de fréquence...

Revenons un peu sur les *buffered synchronous pipelines*...
Mais si : les pipelines fabriqués avec des registres et cadencés par une horloge !

Une sorte de paradoxe

Comme je l'ai dit, une étape d'une instruction s'effectue en un cycle d'horloge. Ainsi, sur un pipeline à n étages, notre instruction mettra n cycles d'horloge à s'exécuter. Par contre, notre instruction s'exécuterait en 1 seul cycle d'horloge sur un processeur sans aucun pipeline. contre un auparavant : on pourrait croire que notre processeur doté de pipeline est n fois plus lent. Mais vu que notre pipeline pourra exécuter n instruction en un cycle d'horloge: on ne perd pas de performances. Visiblement, un processeur utilisant un pipeline de ce genre n'est pas censé être vraiment plus rapide qu'un processeur sans pipeline.

Alors à quoi peut bien servir notre pipeline ?

En fait, il y a anguille sous roche : qui vous a dit que la fréquence était la même ?

Car c'est un fait : posséder un pipeline permet d'augmenter la fréquence du processeur ! Et là, je suis sûr que vous ne voyez pas vraiment le rapport entre le pipeline et la fréquence, et que quelques explications seraient les bienvenues. C'est une histoire de temps de propagation.

Temps de propagation et pipeline

Comme vous le savez tous depuis le second chapitre, un circuit met toujours un certain temps à réagir et à mettre à jour sa sortie quand on modifie son entrée. Plus un circuit contient de portes logiques, plus ce temps sera long : le mot binaire placé sur l'entrée devra "parcourir" plus de portes avant d'arriver sur la sortie (en se faisant modifier au passage). Chaque porte ayant un temps de propagation plus ou moins fixe, plus on traverse de portes, plus cela prend du temps.

La durée d'un cycle du signal d'horloge servant à cadencer un circuit doit absolument être supérieure au temps de propagation

de notre circuit : sans cela, celui-ci ne sera pas synchronisé avec les autres composants et notre processeur ne fonctionnera pas !

Hors, un étage de pipeline est un mini-circuit capable d'effectuer un morceau d'instruction : il contient nettement moins de portes logiques qu'un circuit capable d'exécuter une instruction complète. Donc, un étage de pipeline possède un temps de propagation nettement plus faible qu'un gros circuit capable d'effectuer une instruction rien qu'à lui tout seul ! Vu que le temps de propagation d'un étage est nettement plus petit que le temps de propagation de notre circuit sans pipeline, on peut alors augmenter la fréquence sans risques.

Généralement, plus un processeur possède d'étages, plus il est facile d'augmenter sa fréquence. Certains fabricants de processeurs n'ont d'ailleurs pas hésité à créer des processeurs ayant un nombre d'étages assez élevé dans le but de faire fonctionner leurs processeurs à des fréquences assez élevées. C'est ce qu'a fait Intel avec le Pentium 4. A l'époque, la fréquence d'un processeur était un excellent argument marketing : beaucoup de gens croyaient que plus un processeur avait une fréquence élevée, plus il était puissant. Les fabricants de processeurs cherchaient donc des moyens d'augmenter au maximum la fréquence de leurs processeur de manière sure, pour s'assurer de vendre beaucoup de processeurs. Le pipeline d'un Pentium 4 faisait ainsi entre 20 étages (pour les Pentium 4 basés sur l'architecture Willamette et Northwood), à 31 étages (pour ceux basés sur l'architecture Prescott et Cedar Mill). Pour un exemple plus récent, les processeurs AMD basés sur l'architecture Bulldozer suivent un peu la même approche.

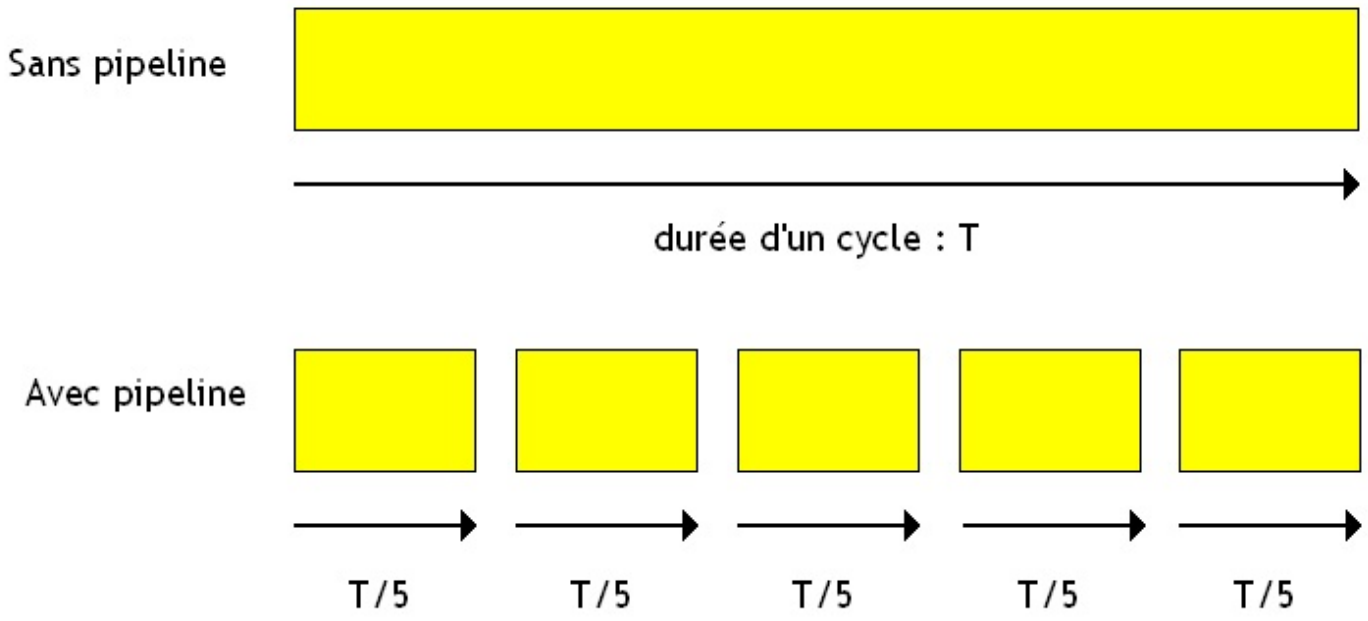
Mais...

On pourrait croire que découper un processeur en x étages permet de multiplier la fréquence par x. Pour vérifier ce qu'il en est, on va vérifier cela avec un raisonnement simple. On va comparer le temps mit par notre instruction pour s'exécuter sur deux processeur identiques à un détail prêt : le premier sera pipeliné, et l'autre non.

Sur notre processeur sans pipeline, une instruction met un temps T pour s'exécuter.

L'autre processeur possède un pipeline de x étages. On notera Tp le temps mit pour exécuter notre instruction sur ce processeur. Pour se simplifier la vie, on va supposer que le découpage des circuits du processeur en étages est très bien fait : chaque étage contient autant de portes logiques que les autres et a le même temps de propagation que ses collègues.

Exemple avec un pipeline à 5 étages.



La durée C d'un cycle d'horloge est donc censée être de :

$$\frac{T}{x}$$

Vu que notre instruction met x cycles pour s'exécuter, on en déduit que le temps d'exécution de notre instruction est donc égal au nombre de cycles nécessaires pour exécuter une instruction $\times$ durée d'un cycle. Ce qui donne :

$$Tp = x \times \frac{T}{x}$$

Ce qui est égal à T, pas de changements.

Mais il faut ajouter un détail : on doit rajouter des registres dans notre pipeline, qui ont eux aussi un temps de propagation qui doit être pris en compte. Un pipeline à x étages possédera $x - 1$ registres intercalés entre ses étages.
Si on note $Tregistre$ le temps de propagation d'un registre, alors le temps mit pour exécuter notre instruction sera de

$$Tp = x \times \frac{T}{x} + Tregistre \times (x - 1)$$

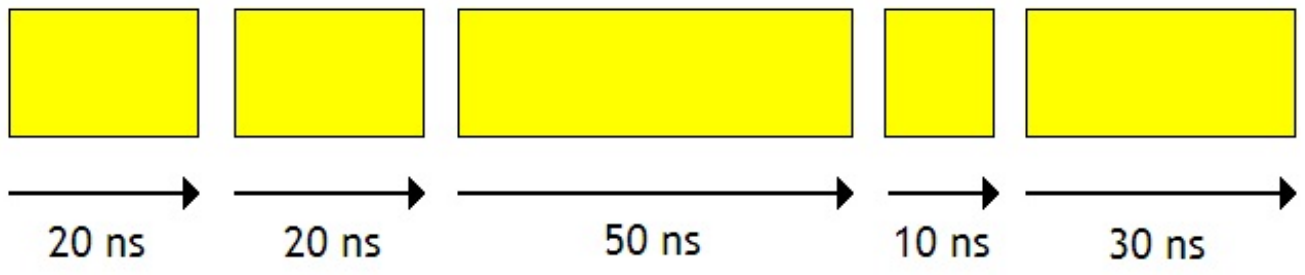
En simplifiant notre équation par x en haut et en bas, on trouve alors que

$$Tp = T + Tregistres \times (x - 1)$$

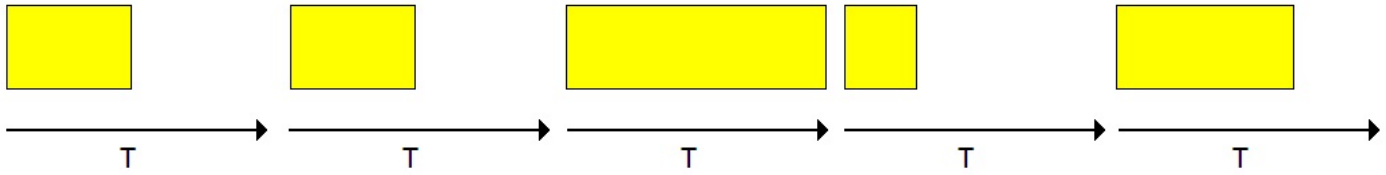
Ce qui est supérieur à T. On voit donc que le pipeline augmente légèrement le temps mit pour exécuter une instruction toute seule. Mais en contrepartie on exécute plusieurs instructions en une fois, ce qui donne un gros avantage au processeur pipeliné.

On peut aussi déduire une autre chose de cette formule : plus un pipeline possède d'étages, plus il faut rajouter de registres entre les étages et plus le temps $Tregistres \times (x - 1)$ augmente. Ainsi, la durée d'une instruction augmente avec le nombre d'étages. Ce qui peut être fortement désavantageux et nuire à la performance de notre processeur.

De plus, il y a aussi un problème avec notre hypothèse de base ; le découpage de notre processeur en étages n'est pas si simple, et certains étages possèdent beaucoup de portes logiques que les autres. Généralement, c'est le découpage du séquenceur qui pose le plus de problème.



La durée d'un cycle d'horloge devra être supérieure au temps de propagation de l'étage le plus fourni en portes logiques. En clair, on se cale sur l'étage le plus lent, ce qui fait augmenter encore plus la durée d'une instruction.



Implémentation hardware

Découper un processeur en étages de pipeline n'est pas une chose facile. Suivant le processeur, le nombre d'étage varie beaucoup, et leur contenu fait de même. Tous nos pipelines ne se ressemblent pas : certains ont un grand nombre d'étapes, d'autres se débrouillent avec peu d'étapes. Aussi, découper un processeur en pipeline peut se faire de différentes manières.

Dans le chapitre sur la micro-architecture d'un processeur, on avait vu que nos instructions machines étaient décomposées en plusieurs micro-opérations qu'on exécutaient les unes pars les autres. Et ce nombre de micro-opérations varie suivant l'instruction. Suivant le pipeline, on peut pipeliner l'exécution des instructions machines, ou celle des micro-opérations. Nous allons commencer par voir un pipeline simple dans lequel toutes nos instructions s'exécuteront en une seule micro-opération. En conséquence, toutes les instructions ont le même nombre d'étages. En conséquence, certains étages sont inutiles pour certaines instructions : vous allez voir de quoi je parle dans ce qui suit. On verra plus tard que les processeurs actuels fonctionnent

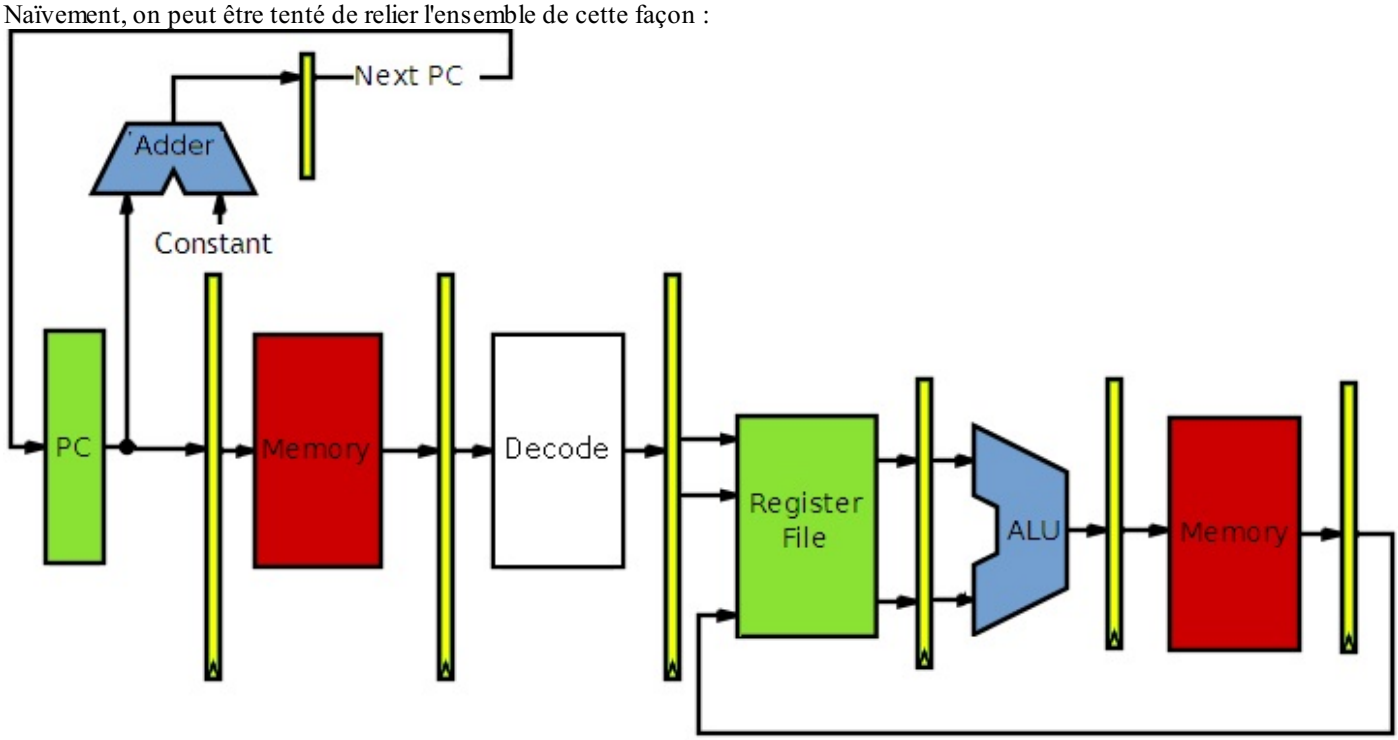
autrement : ils ne pipelinent pas l'exécution des instructions machines, et vont pipeliner l'exécution des micro-opérations à la place. Mais passons.

Pipeline à 7 étages

Nous allons prendre comme exemple un pipeline simple, composé de 7 étages :

- **PC** : mise à jour du Program Counter
- **Fetch** : chargement de l'instruction depuis la mémoire ;
- **Decode** : décodage de l'instruction ;
- **Register Read** : lecture des opérandes dans les registres ;
- **Exec** : calcul impliquant l'ALU ;
- **MEM** : accès mémoire en lecture ou écriture ;
- et **Writeback** : écriture du résultat d'une lecture ou d'un calcul dans les registres.

Avec cette description, on sait grosso-modo quoi mettre dans chaque étage. L'étage de PC va ainsi contenir le *Program Counter*. L'étage de Fetch va devoir utiliser l'interface de communication avec la mémoire. L'étage de Decode contiendra l'unité de décodage d'instruction. L'étage de Register Read contiendra le Register File. L'étage d'Exec contiendra l'ALU, l'étage de MEM aura besoin de l'interface avec la mémoire, et l'étage de Writeback aura besoin des ports d'écriture du Register File.
Naïvement, on peut être tenté de relier l'ensemble de cette façon :

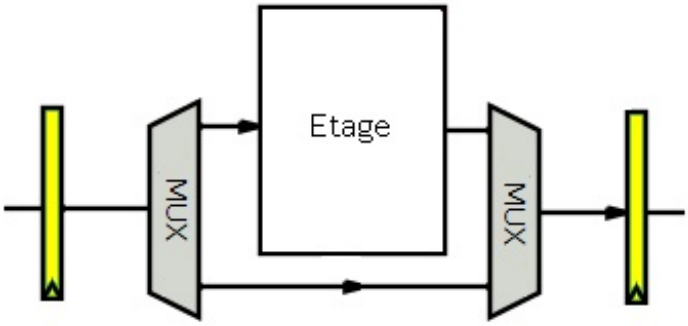


Datapath

Mais cela ne marchera pas ! Toutes les instructions n'ont pas besoin d'accéder à la mémoire, tout comme certaines instructions n'ont pas à utiliser l'ALU ou lire des registres. Certains étages sont "facultatifs" : l'instruction doit quand même passer par ces étages, mais ceux-ci ne doivent rien faire. Par exemple, l'étage MEM ne doit rien faire pour toutes les instructions ne devant pas accéder à la mémoire.

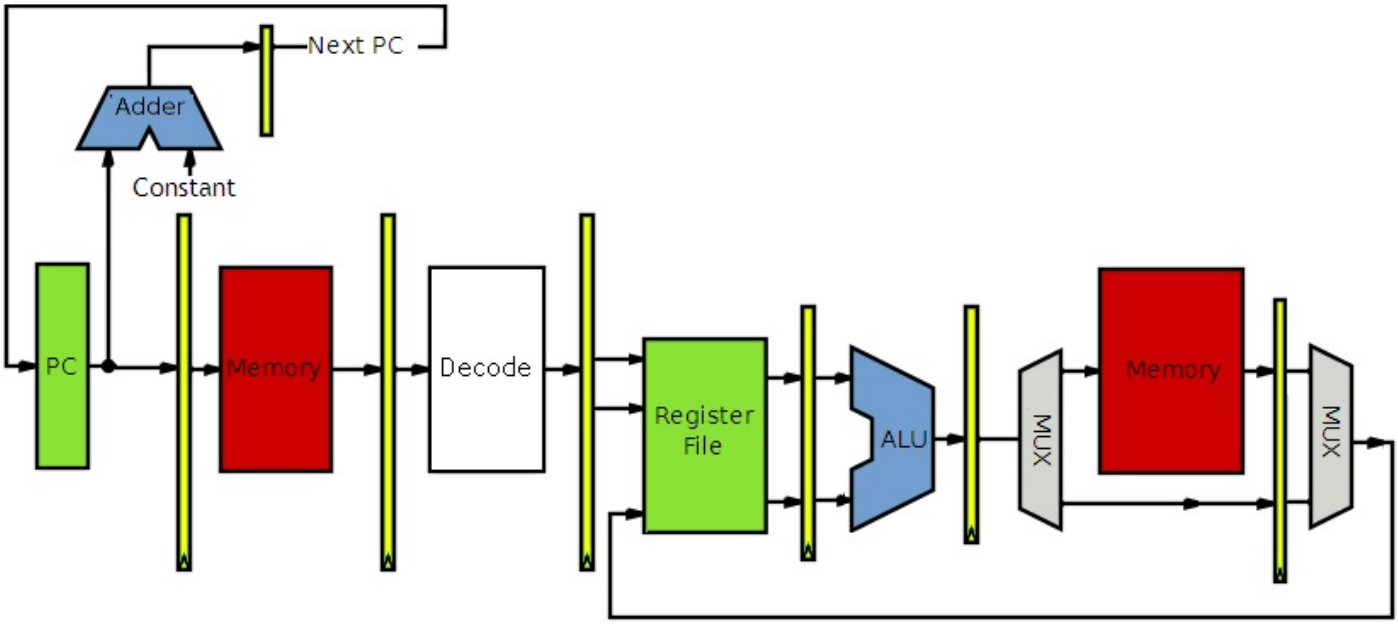
Court-circuit

Il faut donc trouver un moyen pour que nos étages ne fassent rien. Dans le cas où un étage doit être rendu inactif, on peut le court-circuiter en utilisant des multiplexeurs. C'est ainsi : une utilisation intelligente de multiplexeurs peut servir à court-circuiter certains étages du pipeline. Comme vous allez le voir, cela permet de gérer certains modes d'adressage.



Petite remarque : les étages de PC, Fetch et Decode ont toujours quelque chose à faire. Toute instruction doit être chargée et décodée, et le Program Counter mit à jour, que l'on le veuille ou non. Il ne peuvent donc pas être court-circuités.

Tout d'abord, on peut remarquer que certaines instructions n'ont pas besoin d'accéder à la RAM : on doit donc court-circuiter l'étage de MEM. On peut ainsi gérer les instructions comme les additions ou multiplication qui travaillent uniquement dans les registres.

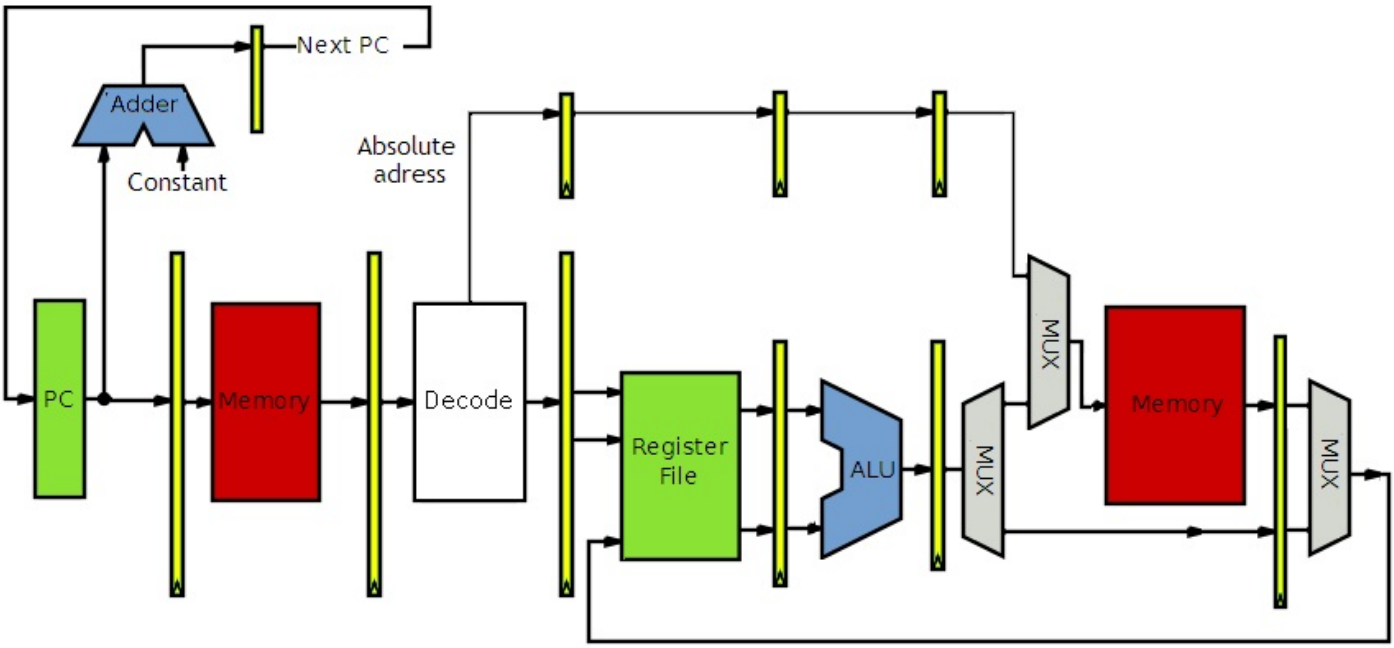


NOP

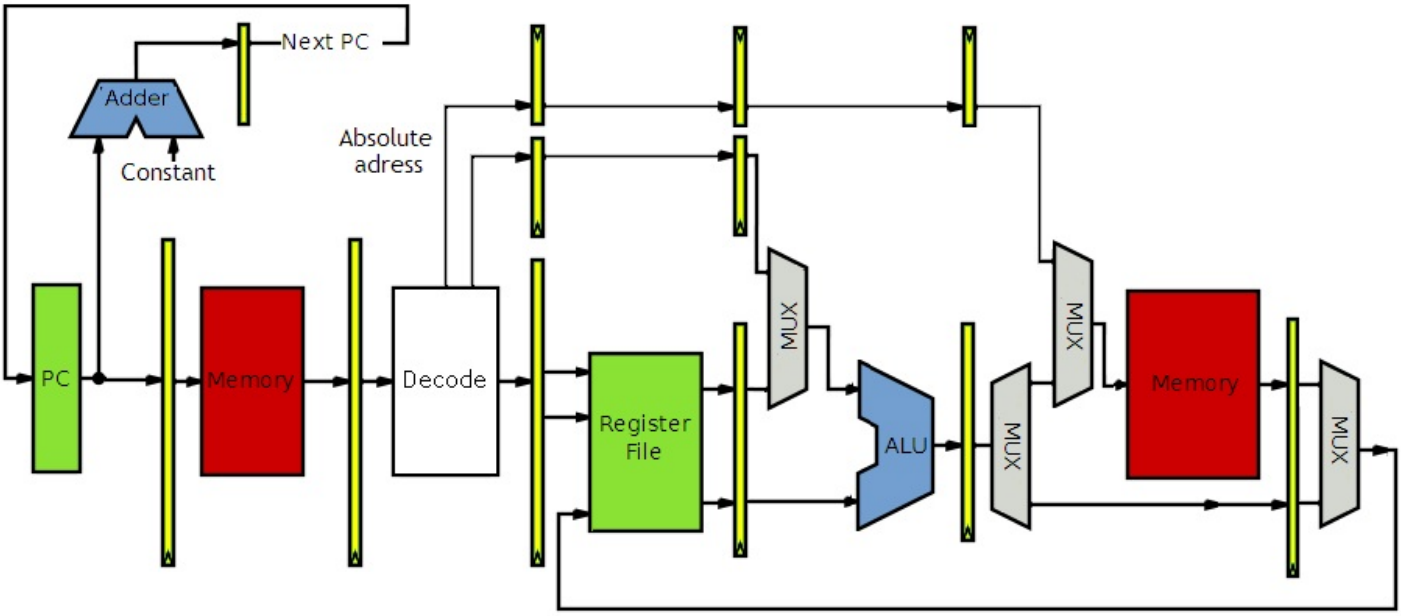
L'ALU aussi doit être court-circuitée dans certaines situations. On n'a pas besoin de l'ALU quand on veut échanger le contenu de deux registres ou qu'on veut envoyer le contenu d'un registre sur le bus d'adresse ou de données. Ceci dit, on n'a pas forcément besoin d'utiliser des multiplexeurs : il suffit juste de faire en sorte que notre ALU puisse effectuer une instruction NOP, à savoir une instruction qui recopie une des entrées de l'ALU sur sa sortie. Même chose pour le *Register File* : il suffit simplement de déconnecter ses entrées et ses sorties : pas besoin de le court-circuiter.

Modes d'adressages

La lecture dans les registres peut être court-circuité lors de l'utilisation de certains modes d'adressage. C'est notamment le cas lors de l'usage du mode d'adressage absolu. Pour le gérer, on doit envoyer l'adresse, fournie par l'unité de décodage, sur l'entrée d'adresse de l'interface de communication avec la mémoire.



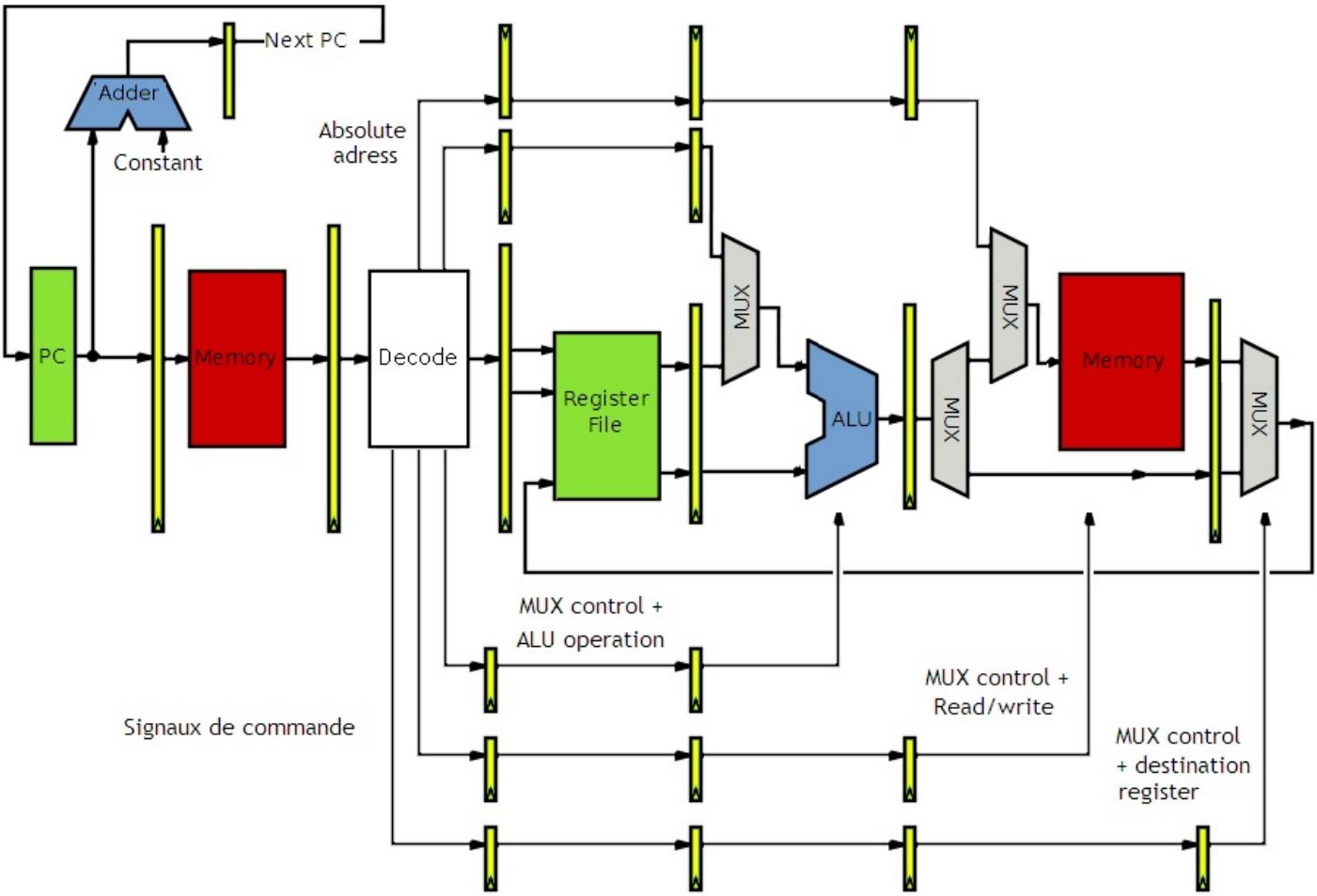
Le principe est le même avec le mode d'adressage immédiat, sauf que l'on envoie une constante sur une entrée de l'ALU.



Signaux de commandes

Les MUXs présents dans notre pipeline doivent être commandés correctement. De même, on doit aussi commander l'ALU, et la mémoire, en leur envoyant des signaux de commande. Seulement, ces signaux de commande sont générés par l'unité de décodage, dans la second étage de notre pipeline. Comment faire pour que ces signaux de commande traversent le pipeline et arrivent au bon moment aux MUXs et aux circuits à configurer ? Relier directement les sorties de l'unité de décodage aux circuits incriminés ne marcherait pas : les signaux de commande arriveraient immédiatement aux circuits, sans temps d'attente : on sauterait des étages !

La réponse consiste simplement à faire passer nos signaux de commande d'un étage à l'autre en utilisant des registres, comme pour les entrées-sorties des MUX et autres unités.

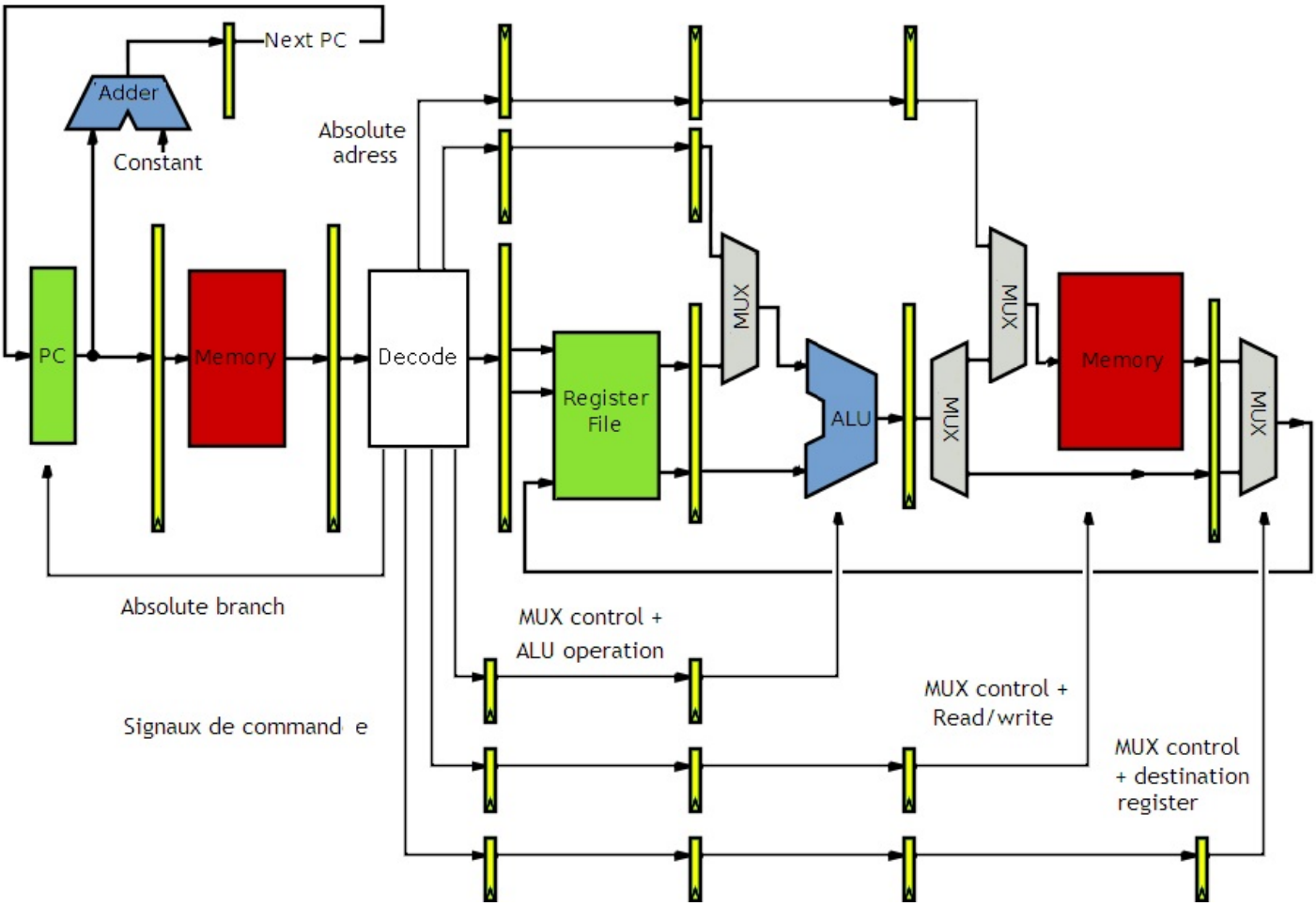


Branchements

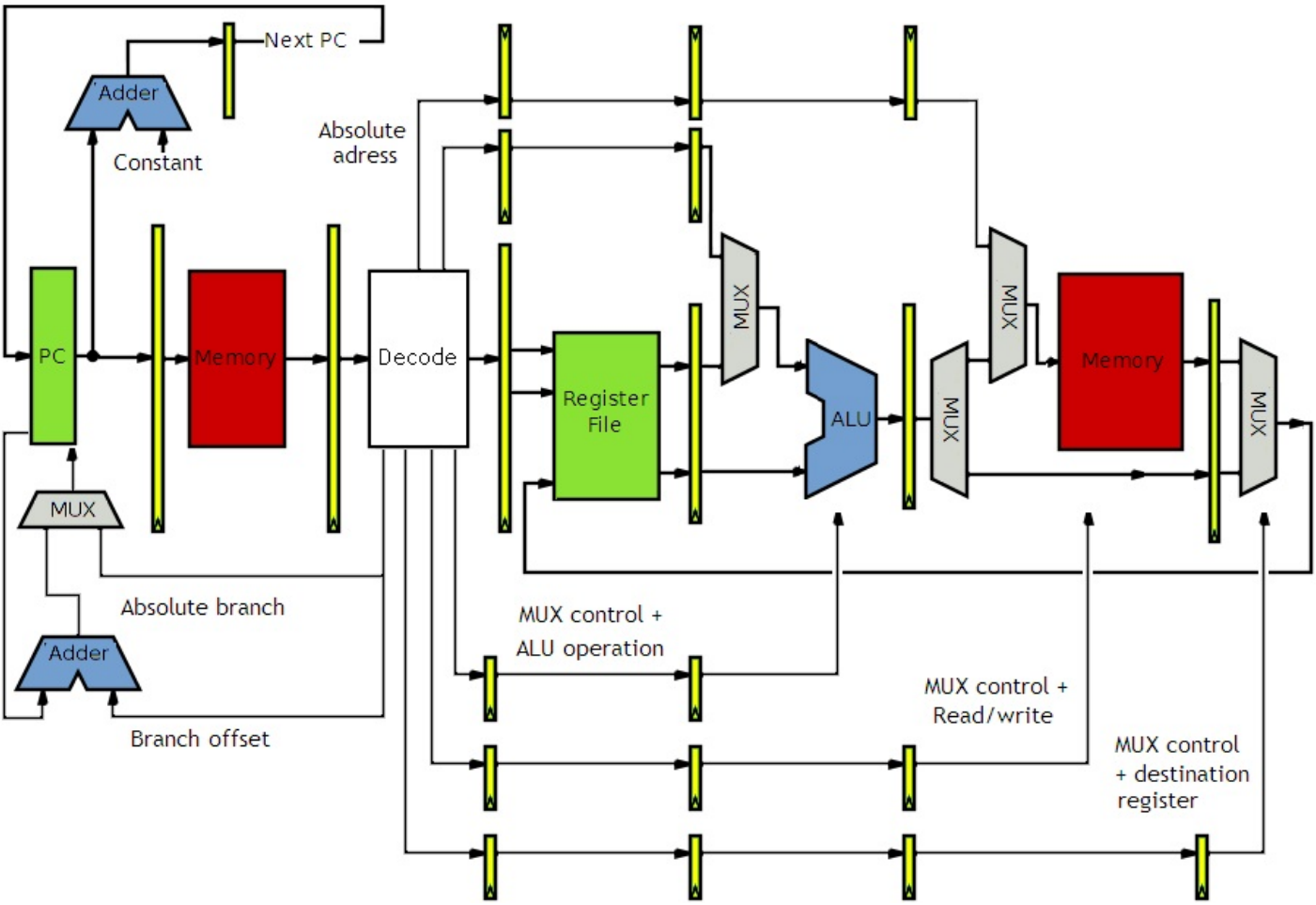
Il ne nous reste plus qu'une dernière catégorie d'instruction à implanter : les branchements.

Splitted branchs

Dans ce qui va suivre, nous allons prendre des instructions de branchements simples, qui ne font que brancher : les comparaisons et branchements sont intégrées dans des instructions à part, et il n'y a pas d'instruction qui fusionne branchement et comparaison. Dans ce cas, rien de bien compliqué : tout dépend du mode d'adressage des branchements. Pour un branchement absolu, il suffit que l'unité de décodage d'instruction envoie l'adresse à laquelle brancher directement dans le *Program Counter*.



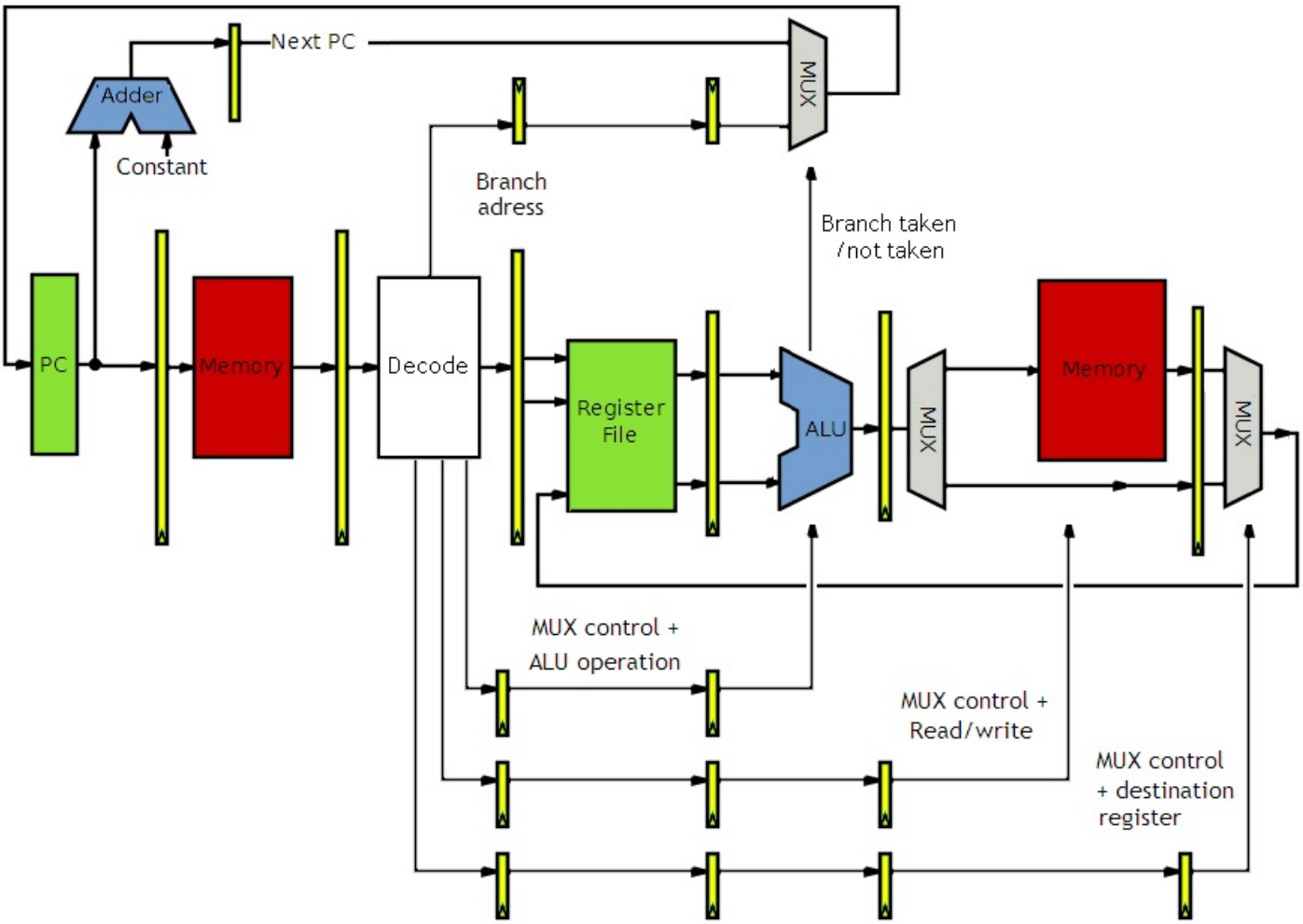
La gestion des branchements relatifs se fait d'une manière assez similaire, en ajoutant des multiplexeurs et un additionneur.



Pour les branchements indirects, on doit relier la sortie du *Register File* au *Program Counter*, en utilisant quelques multiplexeurs, et le tour est joué.

Fused branch

Pour les processeurs qui fusionnent les tests et branchements en une seule instruction, tout change. Cette fois-ci, le choix de l'adresse à laquelle brancher doit s'effectuer une fois le test effectué. Ce test est effectué par l'ALU, et son résultat va servir à commander des MUXs qui détermineront la mise à jour du *Program Counter*.



Pipelines complexes

Avec le pipeline à 7 étages vu plus haut, on fait face à un défaut assez gênant. Avec ce genre de pipeline, toutes les instructions ont exactement le même nombre d'étages. Cela a de gros avantages, avec notamment une gestion simplifiée du pipeline, et une conception plus simple. Mais cela a aussi quelques désavantages : la gestion des instructions multicycles est assez compliquée. Voyons pourquoi.

Micro-opérations

Si vous regardez bien, vous verrez que certains étages sont inutiles pour certaines instructions. Par exemple, si je prends une instruction qui effectue une addition entre deux registres, un des étages ne servira à rien : l'étage MEM. Normal, notre instruction n'accédera pas à la mémoire. Et on peut trouver beaucoup d'exemples de ce type. Par exemple, si je prends une instruction qui copie le contenu d'un registre dans un autre, aie-je besoin de l'étage d'Exec ou de MEM ? Non ! En clair : c'est un peu du gâchis. Si on regarde bien, on s'aperçoit que ce problème de nombre de micro-opérations variable vient du fait qu'il existe diverses classes d'instructions, qui ont chacune des besoins différents.

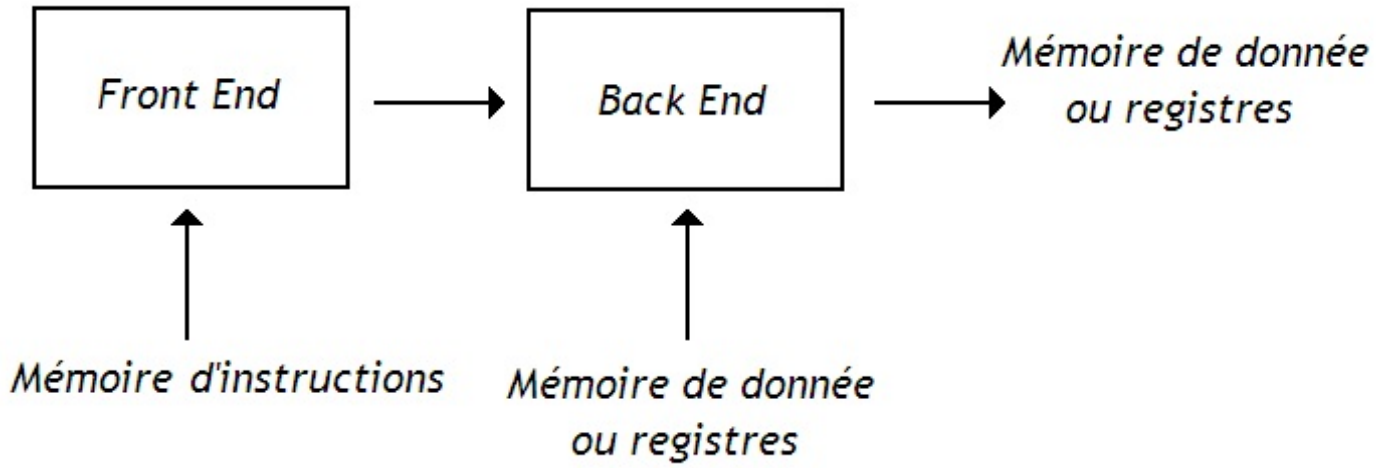
Sur un processeur non pipeliné, on aurait pu éviter de passer par ces étages inutiles. Si vous vous rappelez bien, un processeur normal va découper notre instruction machine en micro-opérations, qu'il exécutera dans un ordre bien précis. Et le nombre de micro-opérations utilisé par une instruction peut varier sans aucun problème : une instruction prendra alors juste ce qu'il faut de micro-opérations pour s'exécuter. Mais avec notre pipeline de longueur fixe, ce n'est pas possible : toutes les instructions utilisent un nombre identique de micro-opérations, chaque micro-opération étant un des étages du pipeline situé après l'étage de ***Decode***.

Principe

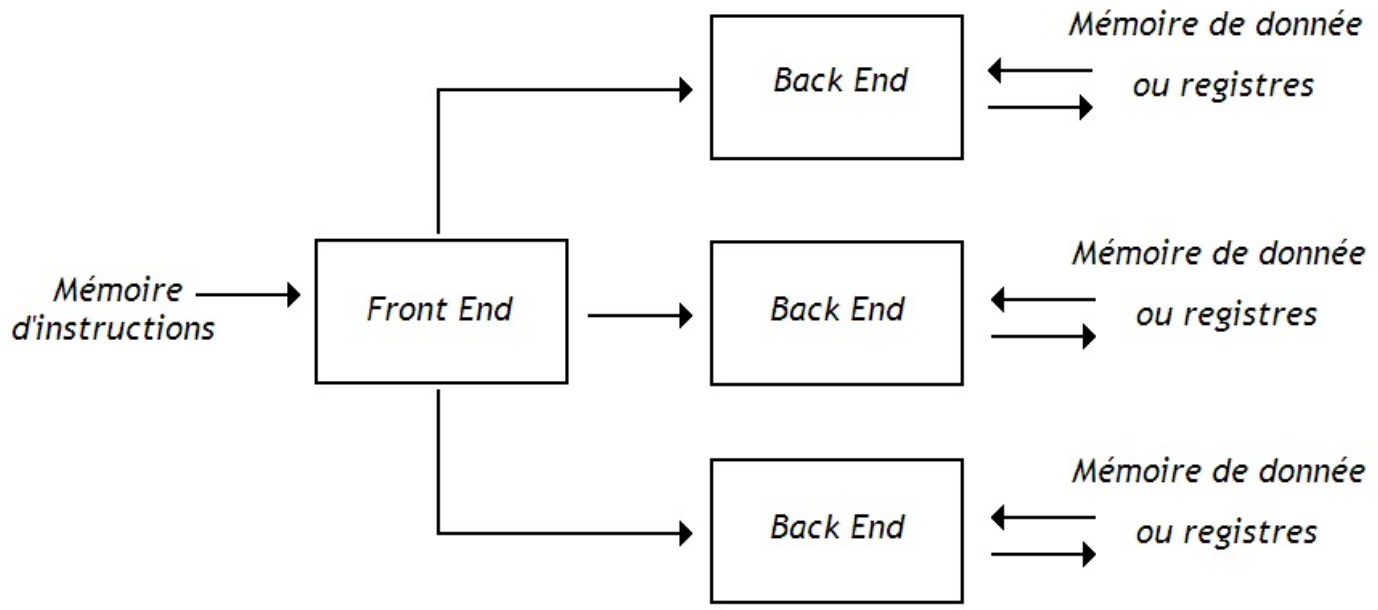
Pour résoudre ce problème, il suffit d'avoir une petite idée : faire varier la longueur du pipeline suivant l'instruction. Si on remarque bien, certains étages sont communs à toutes les instructions : la mise à jour du Program Counter, le Fetch, l'étage de Décodage, etc. Mais pour les autres étages, il arrivent que ceux-ci soient facultatifs dans certaines instructions. Dans ce cas, pourquoi ne pas placer ces étages facultatifs en parallèle les uns des autres, et partager les étages communs ?

Avec cette technique, le pipeline de notre processeur est décomposé en plusieurs parties. La première partie est celle qui est commune à toute les instructions : elle s'occupe de calculer l'adresse de la prochaine instruction, de la charger depuis la mémoire, de la décoder, et parfois plus. Cette partie fonctionne de manière identique pour toutes les instructions. On l'appelle le ***Front***

End. Ce *Front End* est ensuite suivi par une partie du pipeline que l'on nomme le *Back End*.



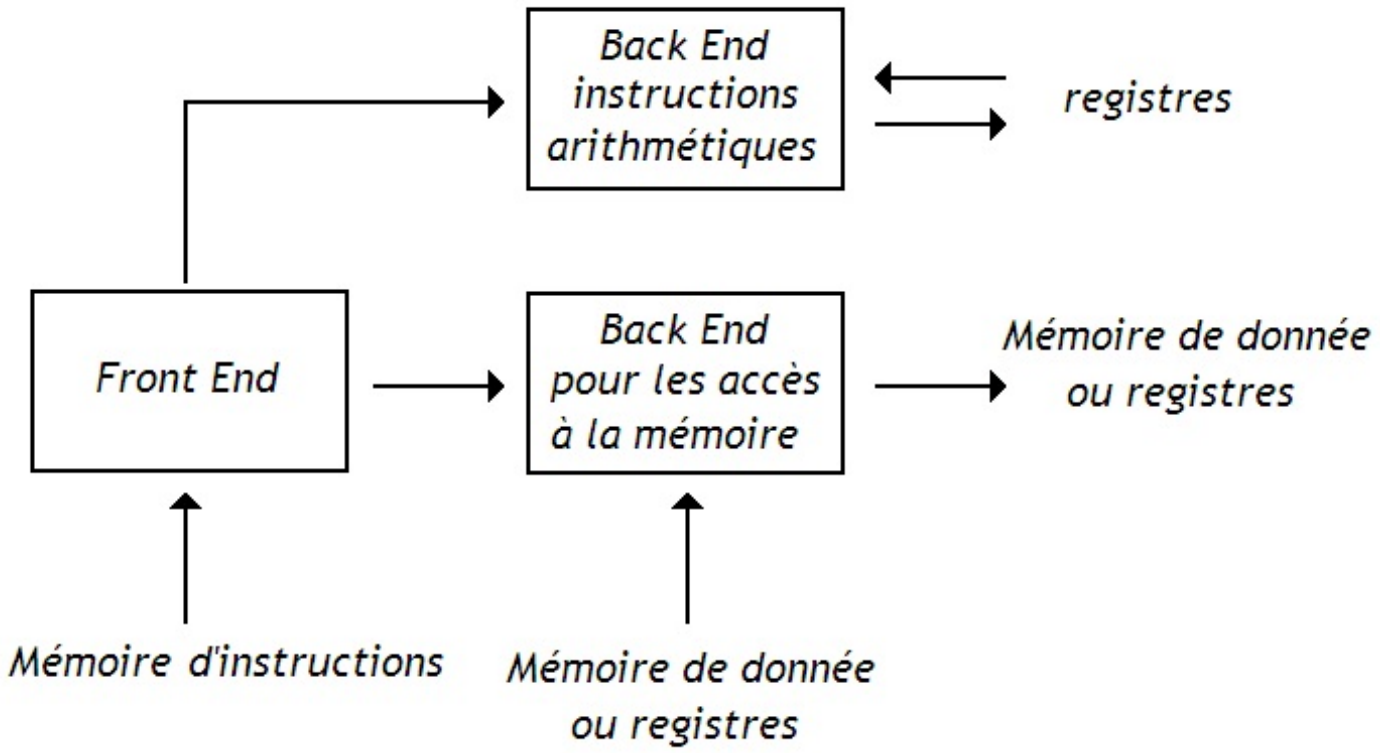
Ce *Back End* est lui-même décomposé en plusieurs unités, chacune adaptée à un certain type d'instruction. Ainsi, une partie du *Back End* sera spécialisé pour les instructions arithmétiques et logiques, qui se passent de l'étape MEM. Une autre partie sera spécialisée dans les opérations d'accès mémoires, qui passent par toutes les étapes du pipeline. Une autre sera réservée aux instructions d'échange de données entre registres, qui se passent des étages Exec et MEM. Et ainsi de suite. On peut considérer que chaque type d'instruction dispose de son propre *Back End*, spécialisé, et dont la longueur peut varier suivant le type d'instructions.



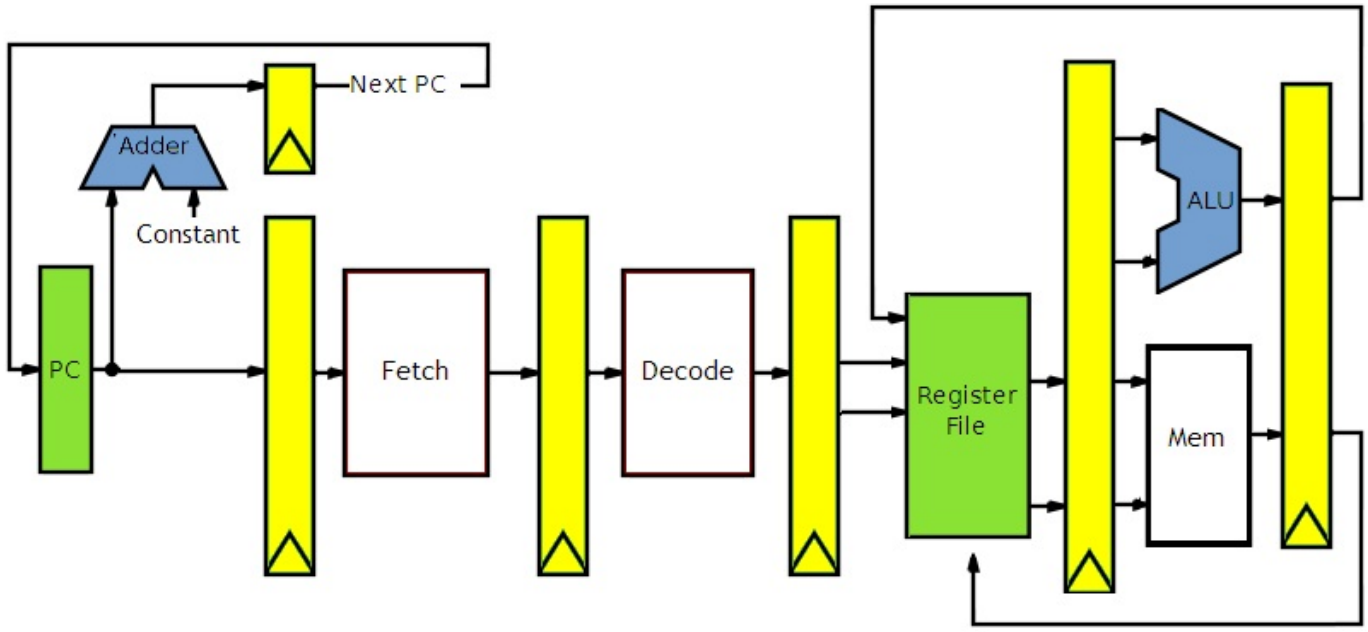
RISC

Le cas le plus simple est celui des processeurs RISC : leur faible nombre de modes d'adressages, et leur simplicité va grandement aider. Grosso-modo, le fait que nos processeurs RISC sont des processeurs de type Load-Store va grandement nous aider : nos processeurs RISC ont des instructions simples, qui ne mélangent pas les accès mémoires avec des opérations arithmétiques et/ou logiques. Les instructions arithmétiques et logiques sont plus simples : elles n'accèdent pas à la mémoire et vont simplement aller trifouiller les registres. Quand aux instructions d'accès mémoires, elles vont simplement calculer une adresse avant d'aller lire ou écrire dans celle-ci.

Dans ce cas, on peut découper notre *Back End* en deux : une partie du pipeline est réservée aux instructions arithmétiques et logiques, tandis qu'une autre est réservée aux instructions mémoires. Notre processeur a juste à fournir deux grosse unités : une pour les accès mémoire, et une autre pour les calculs. Il suffit alors de les placer en parallèle et de rediriger l'instruction dans l'unité qui la prend en charge.



Le pipeline ressemble alors à ceci :



On se retrouve ainsi avec un pipeline contenant un étage de moins. Le pipeline contient donc les étages suivant :

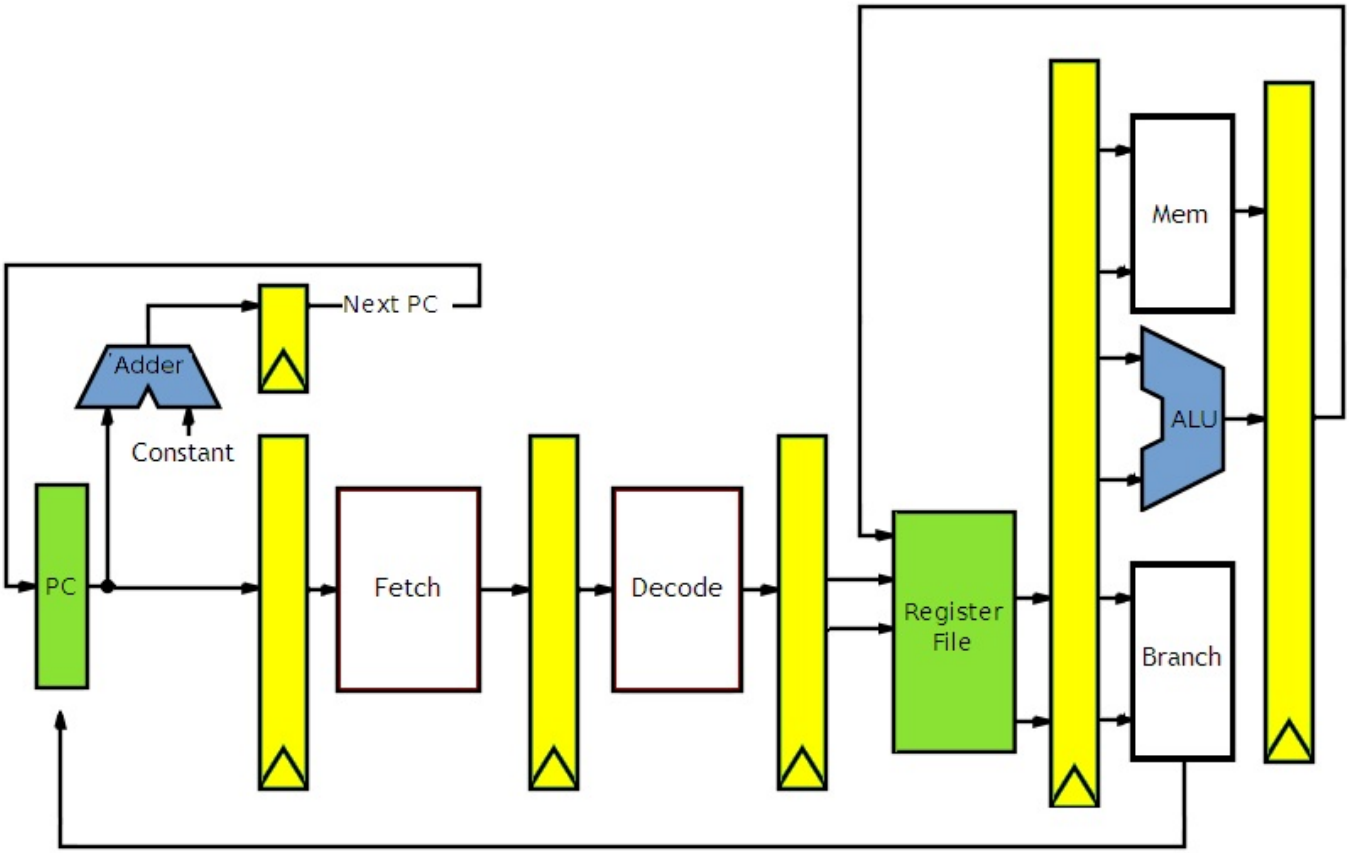
- PC Update ;
- Fetch ;
- Decode ;
- Registre Read ;
- Exec ;
- Write Back.

La ruse vient du fait que l'étage d'Exec est différente suivant chaque instruction. Les instructions arithmétiques s'exécuteront dans leur unité dédiée, et les instructions d'accès mémoire auront leur propre unité spécialisée dans les lectures et écritures. On a

en quelque sorte fusionné l'étage MEM avec l'étage D'Exec.

On peut aussi aller plus loin : l'unité chargée de la gestion des instructions mémoire est souvent découpée en deux sous-unités : une spécialisée dans les écritures, et une autre dans les lectures. Cela permet d'économiser un dernier étage pour les écriture : l'étage de WriteBack, chargé d'enregistrer des données dans les registres. Vous en connaissez beaucoup des écritures qui enregistrent des trucs dans les registres ?

Parfois, on peut effectuer un dernier découpage : on peut séparer la gestion des branchements et celle des instructions arithmétiques et logiques. Les branchements des processeurs RISC sont souvent des branchements qui fusionnent la comparaison avec le branchement proprement dit. Ces branchements vont donc effectuer deux choses : une comparaison, et éventuellement un calcul d'adresse.



Placer des unités en parallèle a un avantage : il est possible d'envoyer des instructions différentes dans des unités séparées en parallèle. C'est possible si l'on dispose d'instructions multicycles, ou que le processeur est prévu pour. On en reparlera dans la suite du tutoriel, mais pour le moment, oubliez ce que je viens de dire. Reste que ce *Back End* peut s'organiser de diverses façons.

CISC

Mais sur les processeurs qui possèdent un grand nombre d'instructions complexes, ou d'instructions avec des modes d'adressages complexes, la situation est assez difficile à résoudre. Il est alors très difficile de faire en sorte que nos instructions prennent juste ce qu'il faut de micro-opérations pour leur exécution. C'est notamment très difficile de pipeliner des processeurs dont certaines instructions peuvent effectuer plusieurs accès mémoires.

Imaginez par exemple, un processeur qui possède une instruction permettant d'additionner deux nombres, et qui va chercher ses opérandes en mémoire, avant d'aller enregistrer son résultat en mémoire. Cette instruction peut être très utile pour additionner le contenu de deux tableaux dans un troisième : il suffit de placer cette instruction dans une boucle, avec quelques autres instructions chargées de gérer l'indice et le tour est joué. Notre instruction se décomposerait alors en beaucoup d'étapes :

- Le calcul de l'adresse de l'instruction (PC update) ;
- le chargement de l'instruction depuis la mémoire d'instruction ;
- le décodage de celle-ci ;
- une étape de calcul d'adresse de la première opérande ;
- le chargement de l'opérande (depuis la mémoire) ;
- une étape de calcul d'adresse de la seconde opérande ;

- le chargement de l'opérande ;
- l'exécution de l'addition ;
- le calcul de l'adresse du résultat ;
- l'enregistrement du résultat en mémoire.

On se retrouverait alors avec un pipeline composé de 10 étages minimum. Dans la réalité cela pourrait aller plus loin. Mais une grande partie de ces étages ne servirait que pour les quelques rares instructions devant effectuer plusieurs accès mémoires. De plus, certains étages seraient identiques : avoir trois étages chargés de calculer des adresses, c'est de la duplication de circuits un peu inutile.

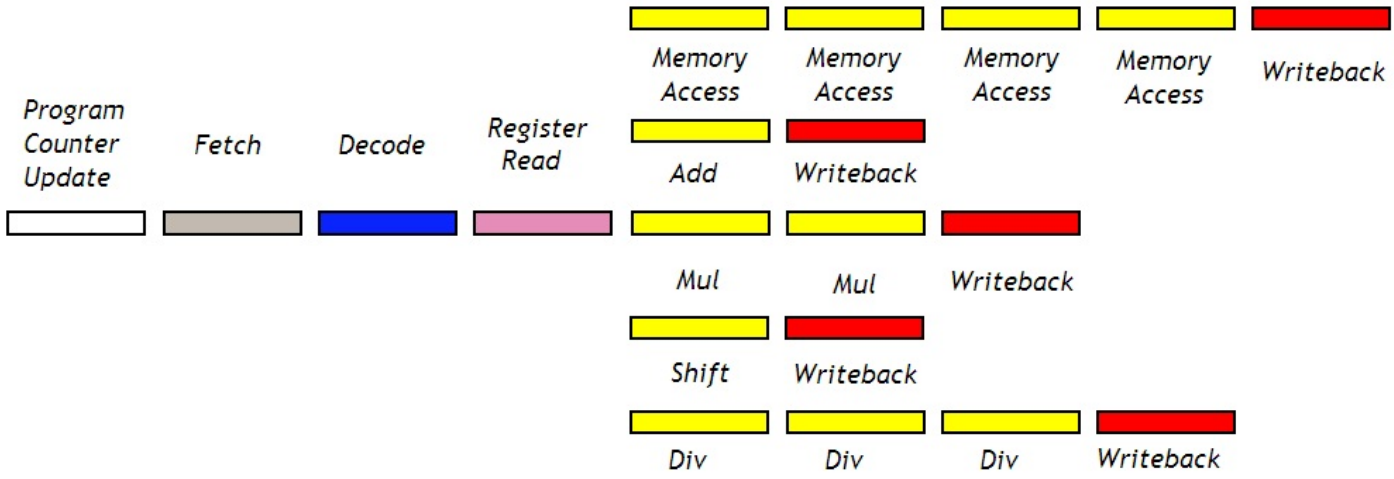
La seule solution, c'est de découper nos instructions machines en sous-instructions qui seront alors chargées dans notre pipeline. Pour faire simple, notre instruction machine est chargée depuis la mémoire décodée, et transformée par le décodeur d'instructions en une suite de micro-instructions, qui sont exécutées chacune dans un ordre bien précis. Ces micro-instructions sont directement exécutables par notre pipeline. Il va de soit que cette organisation complique pas mal le fonctionnement du séquenceur et du pipeline. Par exemple, l'instruction mentionnée au-dessus serait découpée en deux lecture, une instruction d'addition, et une écriture. Cela permet d'utiliser le pipeline RISC vue au-dessus.

Instructions multicyles

Nous avons donc réglé le cas des instructions multicycles dont le nombre de micro-opérations était variable. Il suffit simplement d'envoyer ces instructions dans des unités séparées suivant leurs besoins. Mais il nous reste un cas assez particulier d'instructions multicycles à régler.

Avec un pipeline comme le pipeline à 7 étages qu'on a vu au-dessus, chaque étage dispose de seulement un cycle d'horloge pour faire son travail, et pas plus. Par exemple, toutes les opérations arithmétiques et logiques doivent se faire en un seul cycle. Le temps de propagation des circuits de l'ALU va devoir se caler sur l'opération la plus complexe que celle-ci peut effectuer. Autant vous dire qu'en faisant cela sur un processeur qui implémente des opérations compliquée comme la division, on est mort : la fréquence du processeur ne dépassera pas les 20 Mhz, et il vaudrait mieux abandonner notre pipeline rapidement et revenir à un processeur tout ce qu'il y a de plus normal. En faisant cela, dans notre pipeline à 7 étages, toutes les instructions devront donc s'exécuter en 7 cycles d'horloge, sans possibilité d'avoir de rab. Toutes les instructions doivent se caler sur la plus lente d'entre elle.

Pour éviter cela, on peut permettre à nos instructions de prendre un nombre de cycles d'horloge variable pour s'exécuter. Certaines instructions pourront alors prendre 7 cycles, d'autres 9, d'autres 25, etc. Avec cette solution, on se retrouve face à quelques complications techniques, qu'il faudra gérer.



Mais comment implémenter cette amélioration ?

Stalls

La solution la plus simple consiste simplement à faire en sorte que notre instruction multicycle occupe d'unité de calcul durant plusieurs cycles. Si jamais une instruction a besoin d'utiliser l'ALU durant 4, 5 cycles, elle reste dans l'étage d'Exec durant autant de cycles qu'elle en a besoin.

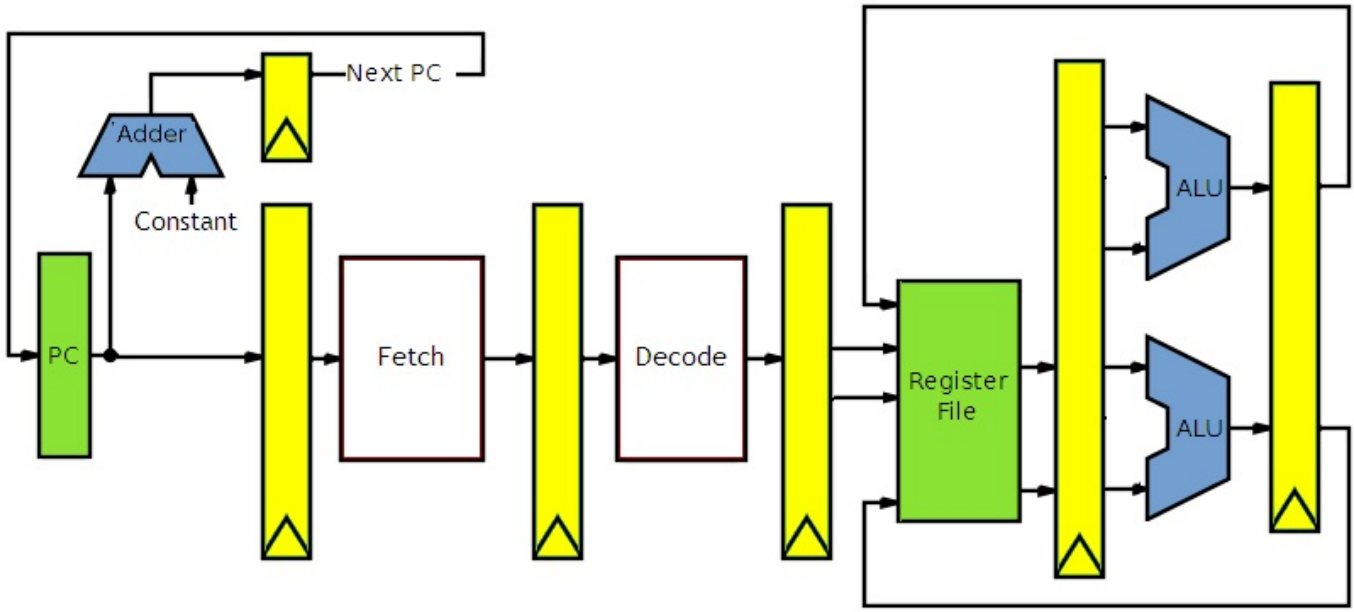
Si j'exécute une instruction multi-cycle qui monopolise l'ALU, il est évident que l'unité de calcul est inutilisable tant que cette instruction multicycle n'est pas terminée. Elle ne peut pas commencer à exécuter une nouvelle instruction. Si une instruction veut

utiliser la même unité de calcul au même moment, on est face à un problème. On fait alors face à ce qu'on appelle une dépendance structurelle : plusieurs instructions veulent occuper un même circuit en même temps. Ici, le circuit en question est l'ALU.

Pour éviter tout problème, notre unité de décodage est mise au courant du fait que notre ALU est occupée et ne peut pas démarrer de nouvelle instruction. Si jamais une instruction qui utilise l'ALU arrive à la fin du Front End, le processeur va vérifier que l'ALU est libre. Si elle ne l'est pas, la nouvelle instruction va devoir attendre que l'ALU se libère. Pour faire attendre cette instruction, on est obligé de bloquer le fonctionnement de certains étages du pipeline. Tout les étages qui précédent l'ALU ne peuvent alors plus rien faire : l'instruction reste bloquée à l'étage de Decode, et toutes les instructions dans les étages précédents sont aussi bloquées. Ce blocage et cette détection des dépendance structurelle est effectuée par quelques circuits implantés dans le processeur. Le blocage proprement dit est assez simple à réaliser : il suffit de ne pas faire parvenir l'horloge aux registres intercalés entre les étages tant que la dépendance n'est pas résolue.

Plusieurs ALUS en parallèles

Pour limiter la casse, on peut disposer de plusieurs unités de calcul en parallèle. Ainsi, on peut dédier certaines unités de calcul pour des opérations lourdes, qui prennent pas mal de cycles d'horloge. A coté, on trouvera des ALU normales, dédiées aux opérations simples. En faisant cela, on évite de bloquer le *Back End* des instructions arithmétiques lors de l'exécution d'une instruction lourde.



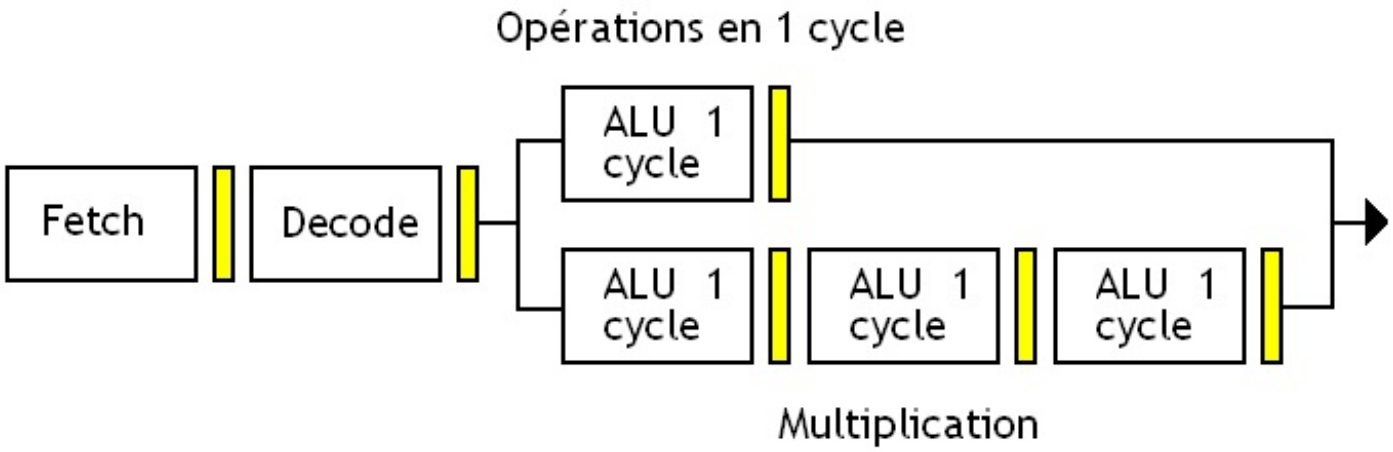
Par exemple, on peut avoir une ALU principale, spécialisée dans les opérations qui durent un cycle d'horloge (additions, soustractions, décalages, etc). Puis, à coté, on peut avoir une ALU pour les multiplications , et une autre pour les divisions. Prenons un exemple : on veut effectuer une instruction de multiplication, qui dure 5 cycles. Sans unités de calcul parallèle, tout le pipeline est bloqué : la seule ALU est occupée par la multiplication. Avec plusieurs ALUs, les autres ALU sont disponibles par effectuer des opérations plus simples : des additions, des soustractions, etc.

En théorie, on peut supprimer toute dépendances structurelles totalement en ajoutant autant d'ALU que notre instruction met de cycles pour s'exécuter. Si on voulait garder des ALU multifonction, il faudrait dupliquer cette grosse ALU en énormément d'exemplaire pour seulement une instruction. Autant dire qu'avec des divisions de 80 cycles, c'est pas la peine d'essayer : c'est trop couteux en circuits. Prenons un exemple : toutes mes instructions arithmétiques prennent un cycle, à part la multiplication qui prend 5 cycles et la division qui prend 40 cycles. Je peux supprimer toute dépendance structurelle en utilisant une ALU pour les opérations en 1 cycle, 5 ALU capable de faire une multiplication, et 40 ALU capables dédiées aux division.

Évidemment, aucun processeur ne fait cela : vous imaginez placer 80 diviseurs dans un processeur, alors que cette opération est une des plus rare qui soit ? Cette technique consistant à dupliquer les ALUs est souvent couplée avec la technique du *Stall*.

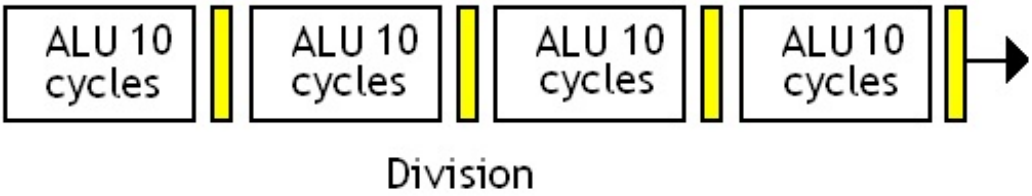
Pipelined ALU

Mais dupliquer des ALU n'est pas la seule solution : on peut aussi pipeliner nos unités de calculs. Si jamais vous avez une opération qui prend 5 cycles, pourquoi ne pas lui fournir un seul circuit, et le pipeliner en 5 étages ? Pour certains circuits, c'est possible. Par exemple, on peut totalement pipeliner une unité de multiplication.



En faisant ainsi, on n'a pas besoin de bloquer notre pipeline : on n'a plus aucune dépendance structurelle. Il est possible de démarrer une nouvelle multiplication à chaque cycle d'horloge dans la même ALU.

Ceci dit, certaines instructions se pipelinent mal, et leur découpage en étages se fait assez mal. Il n'est ainsi pas rare d'avoir à gérer des unités de calcul dont chaque étage peut prendre deux à trois cycles pour s'exécuter. Par exemple, voici ce que cela pourrait donner avec une ALU spécialisée dans les divisions, dont chaque étage fait 10 cycles.

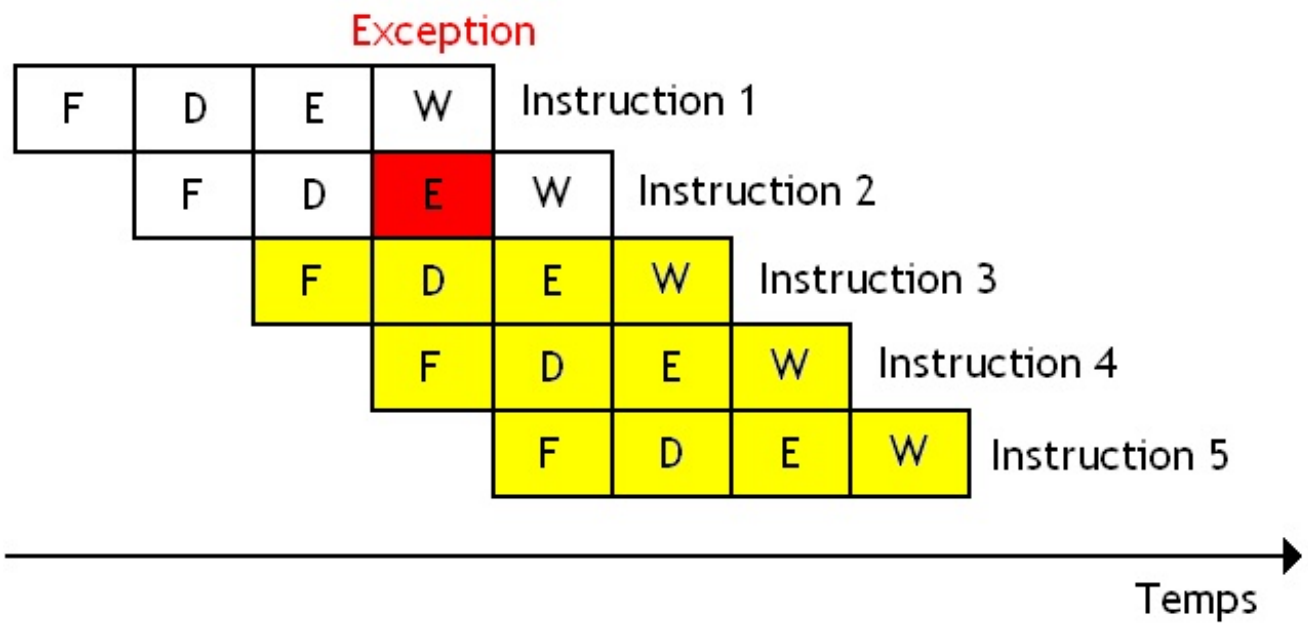


Avec une telle ALU, il faut attendre un certain temps (10 cycles) avant de pouvoir envoyer une nouvelle division dans notre pipeline. Ceci dit, cela ne pose pas souvent de problèmes. Les divisions sont très rares, et il est rare qu'on en lance deux à moins d'une centaine de cycles d'intervalle. Et c'est aussi valable pour d'autres instructions assez rares. Autant dupliquer des ALU pour des instructions courantes est utile, autant dupliquer des unités de division ou des unités spécialisées dans des instructions rares ne sert pas à grand chose : évitons de gaspiller des circuits.

Interruptions et Pipeline

Maintenant, nous devons parler un peu des interruptions et des exceptions. Pour rappel, ces interruptions sont des fonctionnalités de notre processeur qui permettent d'interrompre temporairement l'exécution d'un programme pour éxecuter un morceau de code. Il existe divers types d'interruptions : logicielles, matérielles, et les exceptions. Les interruptions logicielles peuvent être vus comme des appels de sous-programmes spéciaux, tandis que les interruptions matérielles servent à gérer nos périphériques. Quand aux exceptions, elles servent à traiter des erreurs exceptionnelles qui ne devraient pas arriver dans un code bien conçu : un opcode non reconnu, une division par zéro, une erreur de protection mémoire, etc.

Sur un processeur purement séquentiel, ces interruptions et exceptions ne posent aucun problème. Mais si on rajoute un pipeline, les choses changent : nos interruptions et exceptions vont alors jouer les troubles fêtes. Voyons cela par l'exemple. Imaginons que je charge une instruction dans mon pipeline. Celle-ci va alors générer une exception quelques cycles plus tard. Cette exception est indiquée en rouge sur le schéma.



Seul problème, avant que l'exception aie eu lieu, notre processeur aura continué à charger des instructions dans notre pipeline. Il s'agit des instructions en jaune. Et ces instructions sont donc en cours d'exécution dans notre pipeline...alors qu'elles ne devraient pas. En effet, ces instructions sont placées après l'instruction qui est à l'origine de notre exception dans l'ordre du programme. Logiquement, elles n'auraient pas dues être exécutées, vu que l'exception est censée avoir fait brancher notre processeur autre part.

Que faire de ces instructions exécutées trop précocement ? Dans ce cas, il y a deux solutions : soit on s'en moque, et on laisse ces instructions finir leur exécution, soit on trouve un moyen pour que ces instructions ne fassent rien. Dans la première solution, on se moque de ce problème, et on ne prend aucune mesure. Tout se passe comme si l'exception avait eu lieu avec du retard : on dit que l'exception est **imprécise**. Certains processeurs se contentent d'utiliser des interruptions et exceptions imprécises : certains processeurs MIPS ou ARM sont dans ce cas. Mais cela pose quelques problèmes pour les programmeurs, qui doivent faire avec ces interruptions décalées. Autant dans la majorité des cas cela ne pose pas de problèmes, autant cela peut devenir un véritable enfer dans certaines situations.

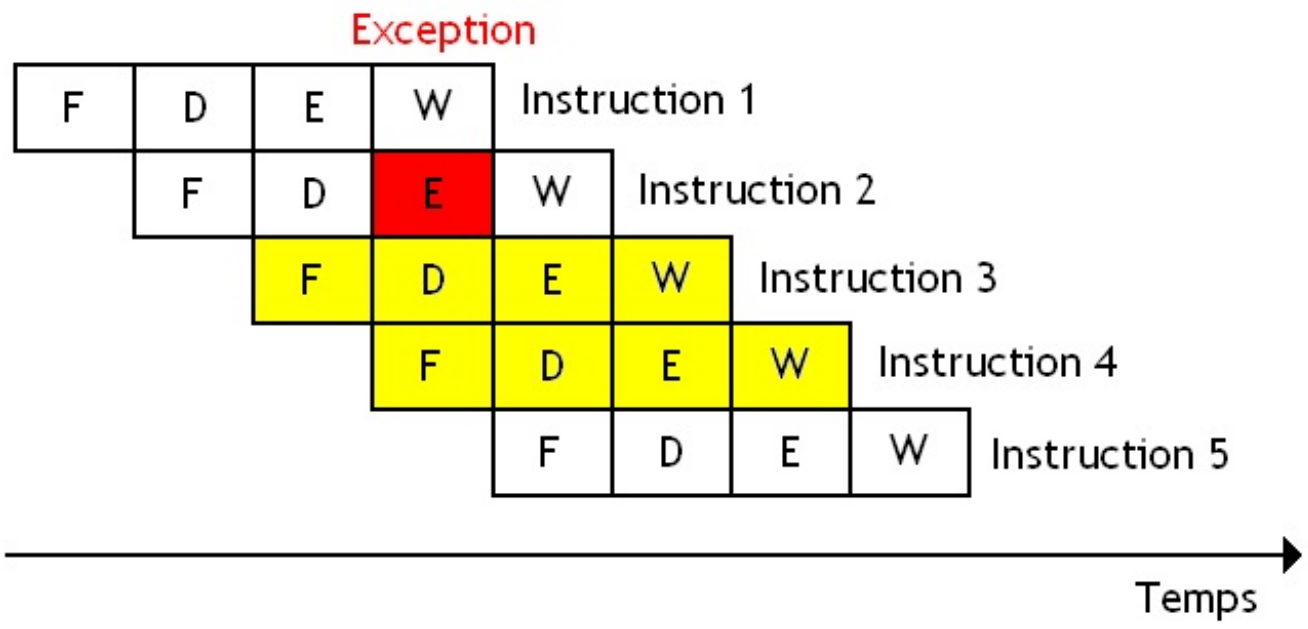
Pour faciliter la vie des programmeurs, certains processeurs ont décidés de supporter des **interruptions précises**. Par interruptions précises, on veut dire que ces interruptions ne se font pas après un temps de retard. Lorsque ces interruptions s'exécutent, le processeur fait en sorte que tout se passe bien, en remettant le pipeline en ordre. Pour ce faire, il existe diverses solutions. Certaines de ces solutions ne permettent que de gérer des interruptions logicielles et des exceptions précises, mais se foutent des interruptions matérielles. D'autres de ces techniques sont plus polyvalentes et gèrent tous les types d'interruptions de façon précise.

NOP Insertion

Pour éviter tout problème, on doit faire en sorte que les instructions chargées en trop ne fassent pas de dégâts dans notre processeur. La solution la plus simple consiste à placer des instructions inutiles après le branchement. Avec ces instructions, le processeur chargera des instructions qui ne feront rien dans notre pipeline, jusqu'à ce que l'adresse à laquelle brancher soit connue.

Dans l'exemple du dessus, l'exception est prise en compte avec 2 cycles d'horloge de retard. Il suffit donc de placer deux instructions inutiles juste après l'instruction pouvant causer une exception. Celle-ci est indiquée en jaune dans le schéma qui

suit.



Le seul problème, c'est qu'insérer ces instructions n'est pas trivial. Si on n'en met pas suffisamment, on risque de se retrouver avec des catastrophes. Et si on en met trop, c'est une catastrophe en terme de performances. Et déduire le nombre exact d'instruction inutiles à ajouter nécessite de connaitre le fonctionnement du pipeline en détail. Autant dire que niveau portabilité, c'est pas la joie !

Au lieu d'insérer ces instructions directement dans le programme, les concepteurs de processeur ont décidé de faire en sorte que le processeur se charge lui-même de gérer nos exceptions et interruptions correctement. Et pour cela, ils ont inventés diverses techniques.

In-Order Completion

Pour limiter la casse, d'autres techniques de gestion des exceptions ont étés inventées. Et celles-ci sont spéculatives. Par spéculatives, on veut dire que le processeur va faire des paris, et exécuter des instructions en spéculant quelque chose. Ici, le processeur va spéculer que les instructions qu'il vient de charger ne lèvent aucune exception, et qu'il peut les exécuter sans aucun problème. Si la spéculation tombe juste, le processeur continuera à exécuter ses instructions, et aura évité des *Pipeline Bubbles* inutiles. Mais si jamais cette prédiction se révèle fausse, il va devoir annuler les dommages effectués par cette erreur de spéculation.

Au final, ce pari est souvent gagnant : les exceptions sont quelque chose de rare, tandis que les instructions capables d'en lever son monnaies courantes dans un programme. Beaucoup d'instructions peuvent lever des exceptions, mais elles en lèvent rarement. On peut gagner en performance avec cette technique. Il faut toutefois signaler que cette technique ne marche que pour les exceptions, pas pour les interruptions logicielles. Ces dernières vont toujours créer des vides dans le pipeline. Mais elles sont rares, et cela ne pose pas de problèmes.

Ordre des écritures

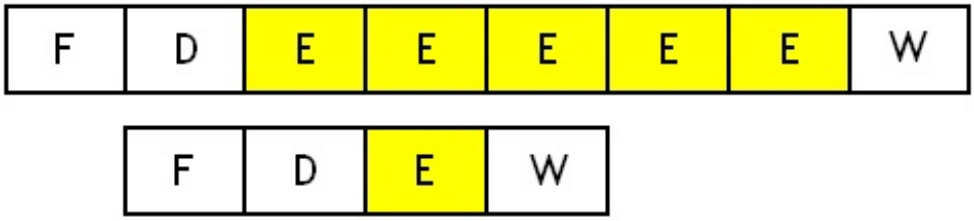
Pour implémenter cette technique, notre processeur doit trouver un moyen pour que les instructions exécutées grâce à la spéculation ne fassent aucun mal si jamais la spéculation est fausse. En clair, toutes les instructions qui suivent une exception dans l'ordre du programme ne doivent pas finir leur exécution, et tous les changements qu'elles peuvent faire doivent être annulés ou empêchés. Ces instructions ne doivent pas aller modifier la mémoire, ou modifier les registres. Et elles ne doivent pas aller toucher au *Program Counter* : sans cela, on ne saura pas à quelle adresse notre programme doit reprendre une fois l'exception traitée. Pour résoudre ce problème, on utilise deux grandes méthodes.

Pour régler ce problème, on doit d'abord faire en sorte que les modifications effectuées par une instruction se fassent dans l'ordre du programme. En clair : les écritures dans la mémoire, les registres, et le *Program Counter* doivent se faire dans le même ordre que celui qui est spécifié dans le programme. On pourrait croire que c'est déjà le cas avec notre pipeline tel qu'il est actuellement : les instructions sont chargées dedans, et s'exécutent dans l'ordre imposé par le programme. Mais en fait, ce n'est pas le cas. La raison est simple : les instructions ne prennent pas le même nombre de cycles, certaines étant plus rapides que d'autres.

Exemple

Prenons cet exemple : je charge deux instructions l'une à la suite de l'autre dans mon pipeline. La première est une lecture en

mémoire et prend 8 cycles pour s'exécuter, tandis que la seconde est une multiplication qui prend 4 cycles.

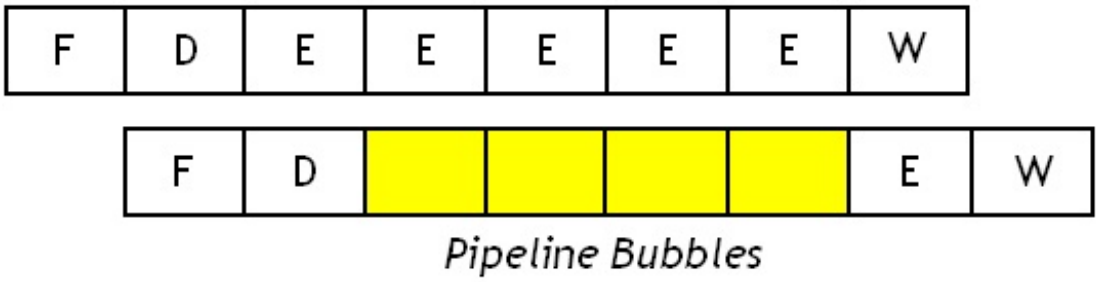


Ces deux instructions sont indépendantes, et pourtant, on est face à un problème. On a beau avoir démarré les instructions dans l'ordre, les résultats de ces instructions ne sont pas enregistrées dans les registres ou la mémoire dans l'ordre imposé par le programme. En clair : si la première instruction lève une exception, les résultats de la seconde auront déjà été enregistrés dans les registres. Le mal est fait : tout se passe comme si l'exception avait été retardé d'une instruction. Cela peut donner n'importe quoi.

On est donc face à un problème : comment maintenir l'ordre d'arrivée des résultats dans les registres/la mémoire ?

How to deal with ?

La première solution est la plus simple : elle consiste à ajouter des *Pipeline Bubbles* pour retarder certaines instructions. Si une instruction est trop rapide et risque d'écrire son résultat avant ses prédécesseurs, il suffit simplement de la retarder en ajoutant quelques *Pipeline Bubbles* au bon moment.



Result Shift Register

Nous allons donc voir comment faire pour implémenter la première solution, à savoir : ajouter des *Pipeline Bubbles* pour retarder les instructions fautives. Pour ce faire, on utilise un circuit spécial, qu'on appelle le ***Result Shift Register*** .

Maintien de l'ordre

Dans sa version la plus simple, il s'agit de ce qu'on appelle un registre à décalage. Ce n'est rien de moins qu'un registre un peu spécial, qui décale son contenu vers la droite à chaque cycle d'horloge. J'ose espérer que vous vous souvenez des décalages, vu dans le chapitre sur l'ALU.

Ce registre à décalage contient autant de bits qu'il y a d'étages dans notre pipeline. Chacun de ces bits signifiera que cet étage est utilisé par une instruction démarrée précédemment. A chaque cycle d'horloge , ce registre est décalé d'un cran vers la droite. Cela permet de prendre en compte le fait que chaque instruction progresse d'un étage à chaque cycle d'horloge. Il va de soit que ce décalage est un décalage logique, qui remplit les vides par des zéros.

Lorsque l'unité de décodage veut démarrer l'exécution d'une instruction, celle-ci va alors vérifier le nombre de cycles que va prendre l'instruction pour s'exécuter. Il va alors vérifier le bit numéro i (en partant de la droite) de ce registre. Si il est à 1, cela signifie qu'une autre instruction est déjà en cours pour cet étage. L'instruction va devoir attendre que cet étage soit libre dans l'unité de décodage. En clair : on ajoute des *Pipeline Bubbles*. Si ce bit est à 0, l'unité va alors le placer à 1, ainsi que tous les bits précédents. Cela permet d'éviter qu'une instruction plus courtes aille s'exécuter avant que notre instruction soit finie. Et l'instruction s'exécutera.

Version améliorée

Dans ses versions plus élaborées, ce *Result Shift Register* contient, pour chaque instruction envoyée dans le pipeline, non seulement un bit qui précise si l'étage est occupé, mais aussi des informations sur l'instruction. L'unité de calcul qu'elle utilise, le registre dans lequel elle veut stocker son résultat, et l'adresse de l'instruction. Ces informations permettent de piloter l'écriture

des résultats de nos instructions en mémoire, et sont utile pour d'unité de décodage : cette dernière a besoin d'informations pour savoir s'il faut générer ou non des *Pipeline Bubbles*.

Par exemple, on stocke l'adresse de l'instruction pour remettre le *Program Counter* à sa bonne valeur, si une exception arrive. Et oui, ce *Program Counter* est modifié automatiquement par l'unité de *Fetch* à chaque cycle. Et si une exception arrive, il faut trouver un moyen de remettre le *Program Counter* à la bonne valeur : celle de l'instruction qui a levée l'exception, afin de pouvoir savoir où brancher au retour de l'exception.

On stocke le registre de destination pour savoir dans quel registre cette instruction doit écrire, et aussi pour savoir quels sont les registres utilisés par les instructions en cours dans le pipeline. J'avais dit plus haut que pour l'unité de décodage devait vérifier si l'instruction qu'elle s'apprête à lancer va lire une donnée écrite pas une instruction encore dans le pipeline. Si c'est le cas, l'unité de décodage doit alors insérer des *Pipeline Bubbles*. Et bien la liste des registres écrit pas les instructions présentes dans notre pipeline se trouve dans ce *Result Shift Register*.

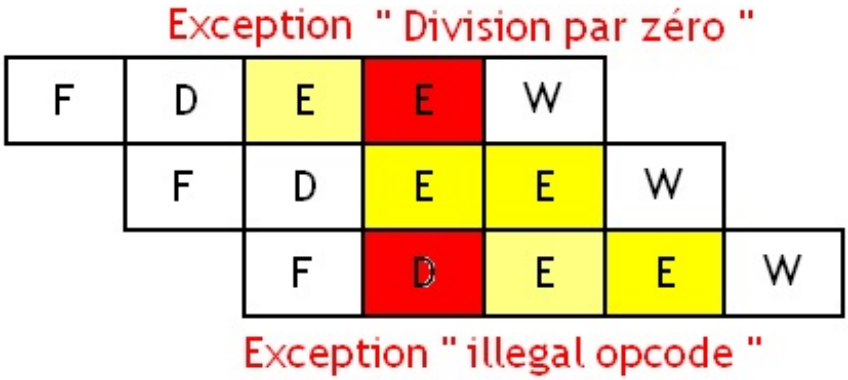
En tout cas, notre *Result Shift Register* est un peu plus complexe. Celui-ci est composé d'entrées, des espèces de blocs de mémoire qui stockent toutes les informations pour une instruction. Unité de calcul utilisée, registre de destination, *Program Counter*, et bit indiquant que l'étage est occupé. Auparavant, avec la version simplifiée du *Result Shift Register*, seul les bits indiquant l'occupation d'un étage étaient décalés vers la droite. Mais avec la version évoluée du *Result Shift Register*, ce sont toutes les entrées qui sont décalées d'un cran à chaque cycle d'horloge.

Speculation Recovery

Maintenant, il nous reste à savoir quand traiter et détecter ces exceptions. Et pour cela, il n'y a pas de miracle : ces exceptions sont détectées dans le pipeline, quand elles sont levées par un circuit. Mais elles ne sont prises en compte qu'au moment d'enregistrer les données en mémoire, dans l'étage de *Writeback*.

Mais pourquoi ?

Imaginez que dans l'exemple du dessus, les deux instructions lèvent une exception à des étages différents. Quelle exception traiter en premier ?

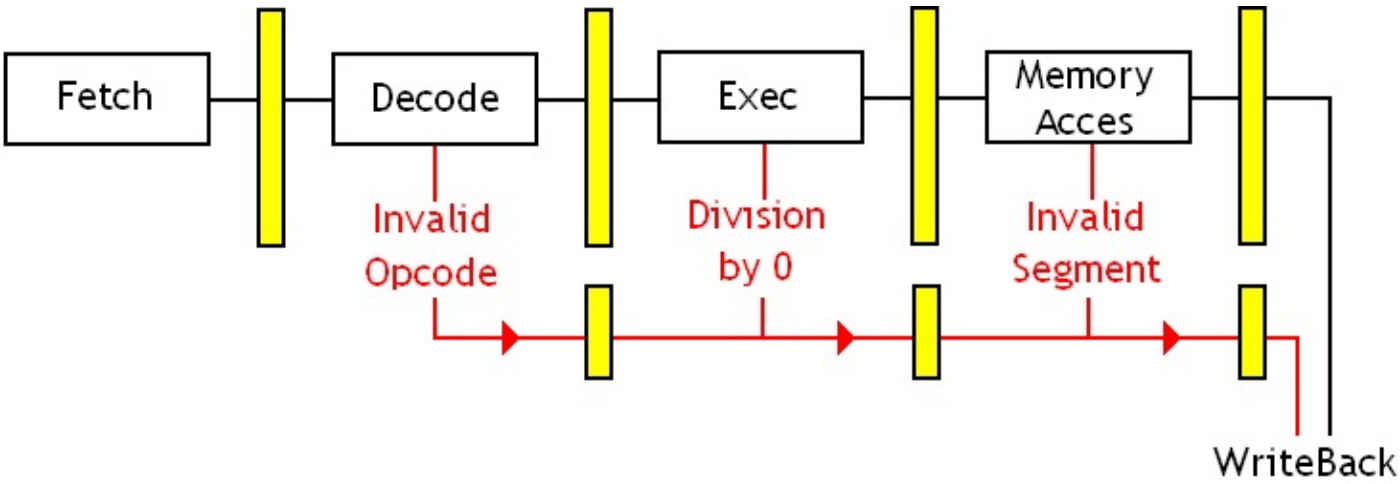


Il va de soit qu'on doit traiter ces exceptions dans l'ordre du programme, donc c'est celle de la première instruction qui doit être traitée. Mais les deux exceptions ont lieu en même temps.

En traitant les exceptions à la fin du pipeline, on permet de traiter les exceptions dans leur ordre d'occurrence dans le programme. C'est très utile quand plusieurs exceptions sont levées dans le pipeline.

Prise en compte des exceptions

Lorsqu'une exception a lieu dans un circuit de notre processeur, ce circuit va devoir prévenir qu'il y a eu une exception. Mais vu que celle-ci doit être prise en compte une fois que l'instruction fautive arrive à la fin du pipeline, il faut trouver un moyen de mettre en attente celle-ci. Pour cela, nos circuits vont disposer de quelques bits, qui indiquent si une exception a eu lieu et qui permettent de préciser laquelle. Ces bits vont ensuite passer d'un étage à un autre, en même temps que l'instruction qui a levée l'exception.



Une fois arrivé à la fin, un petit circuit combinatoire va alors vérifier ces bits (pour voir si une exception a étè levée), et va agir en conséquence.

Annulation des écritures fautives

Maintenant, comment faire pour remettre le pipeline en bon état si une exception arrive ? Après tout, il faut bien éviter que nos instructions exécutées trop précocement, celles qui suivent l'exceptions et ont étés chargées dans notre pipeline, ne fassent pas de dégâts. On doit remettre notre pipeline comme si ces instructions n'avaient pas étés chargées.

La solution est très simple . Il suffit de rajouter un dernier étage dans le pipeline, qui sera chargé d'enregistrer les données dans les registres et la mémoire. Si jamais une exception a lieu, il suffit de ne pas enregistrer les résultats des instructions suivantes dans les registres, jusqu'à ce que toutes les instructions fautives aient quittées le pipeline. Ainsi, ni la mémoire, ni les registres, ni le *Program Counter* ne seront modifiés si une exception est levée. Tout se passera comme si ces instructions exécutées trop précocement ne s'étaient jamais exécutées.

Out Of Order Completion

On a vu plus haut que pour gérer correctement nos exceptions et interruptions précises, nous sommes obligés de maintenir l'ordre des écritures en mémoire ou dans nos registres. Nous avons vu une première solution qui consistait à retarder l'exécution de certaines instructions en ajoutant des *Pipeline Bubbles*. Mais il existe d'autres solutions. Ces solutions évitent de devoir retarder certaines instructions, et évite ainsi d'insérer des *Pipeline Bubbles* dans notre pipeline. Notre pipeline pourra exécuter des instructions à la place : on gagne ainsi en performances. Nous allons voir certaines de ces techniques dans ce qui suit.

Il existe plusieurs solutions à ce problème. On peut utiliser :

- Des techniques de *Register Checkpointing* ;
- Un *Reorder Buffer* ;
- Un *History Buffer* ;
- Un *Future File* ;
- ou autre chose.

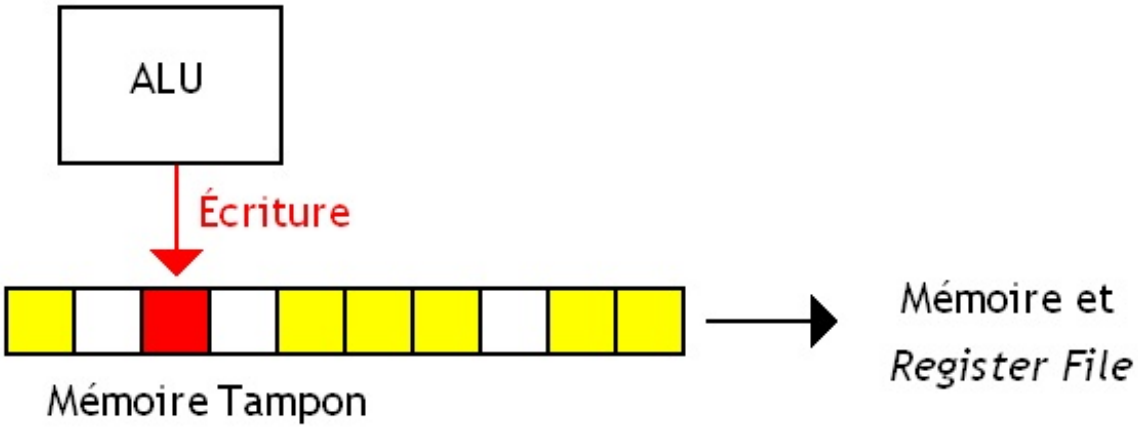
Register Checkpointing

La première technique, le ***Register Checkpointing***, est assez simple. Elle consiste à faire des sauvegardes régulières de nos registres de temps à autre, et récupérer cette sauvegarde lors d'une exception. Il suffirait alors de poursuivre l'exécution à partir de notre sauvegarde jusqu'à retomber sur l'instruction qui lève l'exception, et agir en conséquence.

Seul problème, cette solution est lente : sauvegarder tous les registres du processeur n'est pas gratuit. En conséquence, cette solution n'est donc jamais utilisée.

Re-Order Buffer

La première solution consiste à exécuter nos instructions sans se préoccuper de l'ordre des écritures, avant de remettre celles-ci dans le bon ordre. Le tout sera d'envoyer dans le bon ordre les résultats des instructions dans les registres. Pour remettre en ordre ces écritures, les résultats des instructions seront mis en attente dans une sorte de mémoire tampon, avant d'être autorisés à être enregistrés dans les registres une fois que cela ne pose pas de problèmes.



Cette mémoire tampon s'appelle le ***Reorder Buffer***, et elle fera l'objet d'un chapitre dans la suite du tutoriel. On la laisse de coté pour le moment.

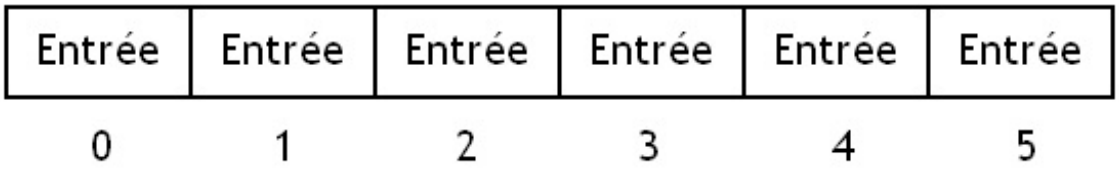
History Buffer

Autre solution pour retrouver les bonnes valeurs de nos registres : on laisse nos instructions écrire dans les registres dans l'ordre qu'elles veulent, mais on garde des informations pour retrouver les bonnes valeurs de nos registres. Ces informations sont stockées dans ce qu'on appelle l'***History buffer***.

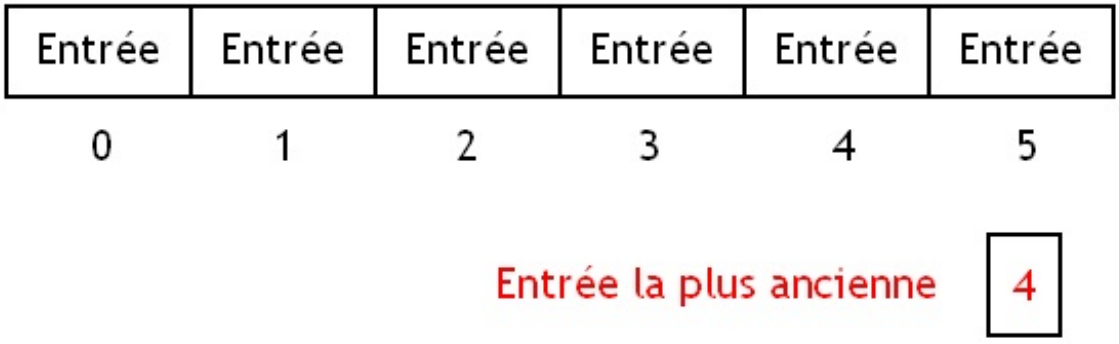
Une file

Cette sauvegarde doit être faite pour chaque instruction. Ainsi, notre *History buffer* est organisé en plusieurs entrées, des espèces de blocs de mémoire qui de quoi revenir à la normale pour chaque instruction. A chaque fois qu'on veut démarrer une instruction, on réserve une entrée pour notre instruction, et on la rempli avec les informations qu'il faut. Une fois qu'on sait que notre instruction s'est terminée sans exception, on libère l'entrée occupée par notre instruction. Pour gérer nos instructions dans l'ordre du programme, ces entrées sont triées de la plus ancienne a la plus récente. Voyons comment cela est fait.

Notre *History buffer* contient un nombre d'entrée qui est fixé, câblé une fois pour toute. Chacune de ces entrées est identifiée par un nombre, qui lui est attribué définitivement.



Ces entrées sont gérées par leur identifiant. Le numéro de l'entrée la plus ancienne est ainsi mémorisé dans une petite mémoire. Cela permet de pointer sur cet entrée, qui contient la prochaine donnée à enregistrer dans les registres ou la mémoire.



Quand cette entrée quitte le *History buffer*, le numéro, le pointeurs sur la dernière entrée est augmenté de 1, pour pointer sur la prochaine entrée.

De même, le numéro de l'entrée la plus récente est aussi mémorisé dans une petite mémoire. Il faut bien savoir où ajouter de

nouvelles entrées. Ainsi, le *History buffer* sait quelles sont les entrées valides : ce sont celles qui sont situées entre ces deux entrées.

Quand on ajoute une instruction dans le *History buffer*, il ne faut pas oublier d'augmenter ce numéro de 1.

Petit détail : quand on ajoute des instructions dans le *History buffer*, il se peut que l'on arrive au bout, à l'entrée qui a le plus grand nombre. Pourtant, le *History buffer* n'est pas plein. De la place se libère dans les entrées basses, au fur et à mesure que le *History buffer*. Dans ce cas, on n'hésite pas à reprendre depuis le début.

Exemple : je prends l'exemple précédent, avec l'entrée 4 occupée. Si je rajoute une instruction, je remplirais l'entrée 5.

La prochaine entrée à être remplie sera l'entrée numéroté 0. Et on poursuivra ainsi de suite.

Contenu des entrées

Lorsqu'une instruction s'exécute, elle va souvent modifier le contenu d'un registre. Si jamais notre instruction a étè exécutée alors qu'elle n'aurait pas due, on perdra l'ancienne valeur de ce registre et on ne pourra pas la remettre à la normale. La solution est de sauvegarder cette ancienne valeur dans notre *History buffer*. Ainsi, quand on envoie notre instruction à l'ALU, on va lire le contenu du registre de destination, et le sauver dans l'*History buffer*. Ainsi, en cas d'exception, on sait retrouver la valeur modifiée par notre instruction.

Mais cette valeur, dans quel registre la remettre en place ? Et bien on n'a pas trop le choix : on doit aussi se souvenir quel registre à étè modifié par notre instruction.

Ensuite, il faut savoir si l'exécution de l'instruction est terminée. On va donc rajouter un bit ***Valid***, qui indique si l'instruction est terminée.

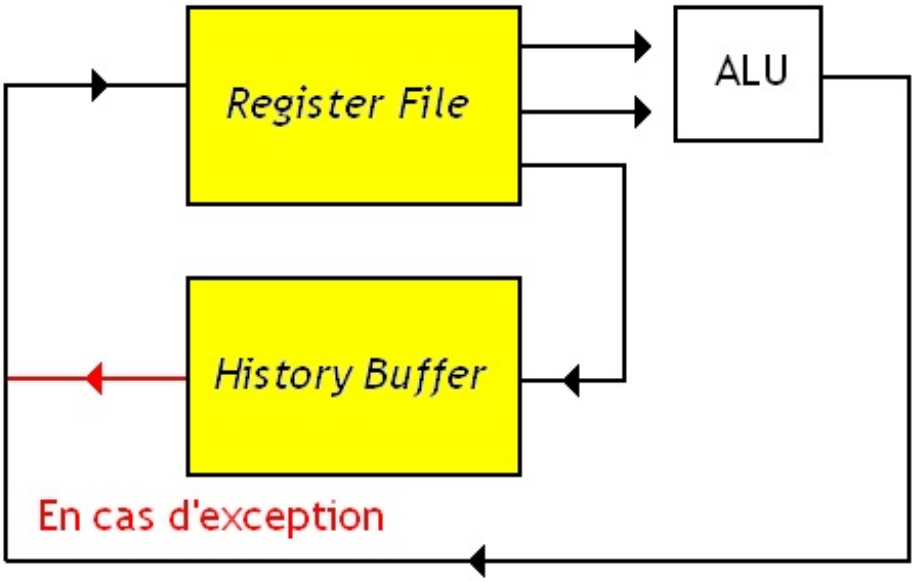
De même, il faut savoir si celle-ci s'est terminée sans exception. Et on rajoute un bit ***Exception*** qui indique si l'instruction a levée une exception. Avec cette technique, les exceptions sont encore une fois gérées à la fin de l'instruction : la détection des exceptions se fait au dernier étage de notre pipeline, exactement comme dans les techniques précédentes.

Et enfin, si notre instruction lève une exception, on doit savoir où reprendre. Donc, on sauvegarde l'adresse de notre instruction. Celle-ci est disponible dans le *Program Counter*.

Principe

Lorsqu'une instruction située dans l'entrée la plus ancienne a levé une exception, celle-ci est détectée grâce au bit *Exception* dans l'entrée qu'elle occupe. Il suffit alors d'annuler toutes les modifications faites par cette instruction et les suivantes. Pour annuler les modifications d'une instruction, il suffit de remettre le registre modifié par l'instruction a son ancienne valeur. Pour cela, on utilise les informations stockées dans l'*History buffer*.

Pour remettre les registres dans leur état valide, on vide l'*History buffer* dans l'ordre inverse d'ajout des instruction. En clair : on commence par annuler les effets de l'instruction la plus récente, et on continue de la plus récente à la plus ancienne jusqu'à vider totalement l'*History buffer*. Une fois le tout terminé, on retrouve bien nos registres tels qu'ils étaient avant l'éxecution de l'exception.



Future File

L'*History Buffer* possède un grand défaut : en cas d'exception ou d'interruption, on est obligé de remettre les registres à leur bonne valeur. Et cela se fait en copiant la bonne valeur de l'*History Buffer* dans les registres. Et cela peut prendre du temps. Cela peut très bien ne pas poser de problèmes : après tout, les exceptions sont rares. Seul problème : on verra dans les prochains chapitres que les branchements posent des problèmes similaires aux exceptions et que les solutions utilisées pour nos exceptions sont réutilisables. Mais les branchements sont beaucoup plus nombreux que les exceptions : environ une instruction sur 10 est un branchement. Et dans ces conditions, l'*History Buffer* et ses remises en l'état des registres sont trop lentes. On doit donc trouver une autre alternative.

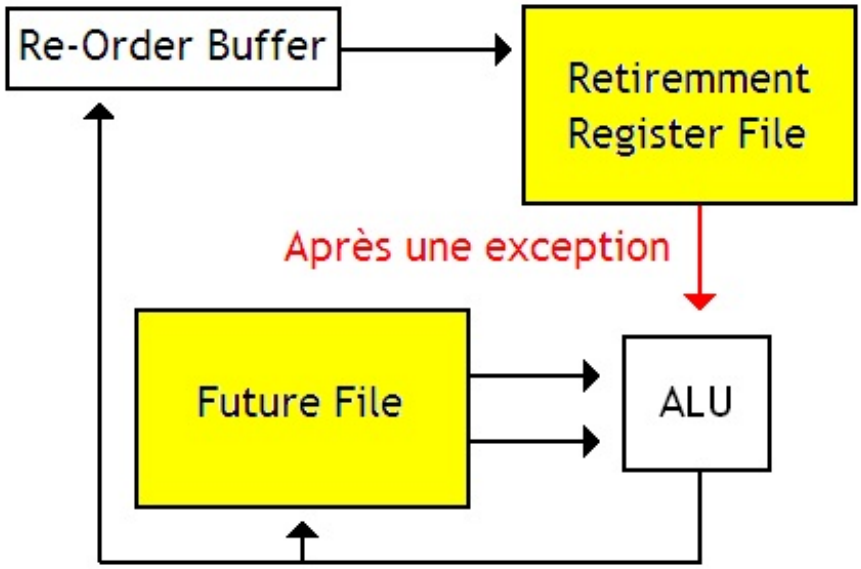
Deux Register Files

La solution est celle du ***Future File***. Cette solution consiste à avoir deux *Register File*. Le premier *Register File* contient les valeurs les plus récentes, celles qu'on obtient en supposant l'absence d'exceptions. On l'appelle le *Future File*. L'autre va servir à stocker les données valides en cas d'exception, et il s'appelle le ***Retiremment Register File***.

Avec cette technique, nos instructions sont exécutées sans se préoccuper de l'ordre de leurs écritures. Leur résultat sera

enregistré dans le *Future File*. Les opérandes de nos instructions seront aussi lues depuis le *Future File*, sauf en cas d'exception. En cas d'exception, les opérandes sont alors lues depuis l'autre *Register File*, celui qui contient les données valides en cas d'exception.

Pour que ce dernier contiennent les bonnes données, on doit lui envoyer les résultats des instructions dans le bon ordre. Pour remettre en ordre ces écritures, les résultats des instructions seront mis en attente dans une sorte de mémoire tampon, avant d'être autorisés à être enregistrés dans les registres une fois que cela ne pose pas de problèmes. Cette mémoire tampon est un *History Buffer* modifié qu'on appelle le ***Re-Order Buffer***. Celui-ci contient exactement la même chose qu'un *History Buffer* normal, à un détail prêt : au lieu de stocker l'ancienne valeur d'un registre (celle avant modification), il stockera le résultat de l'instruction.



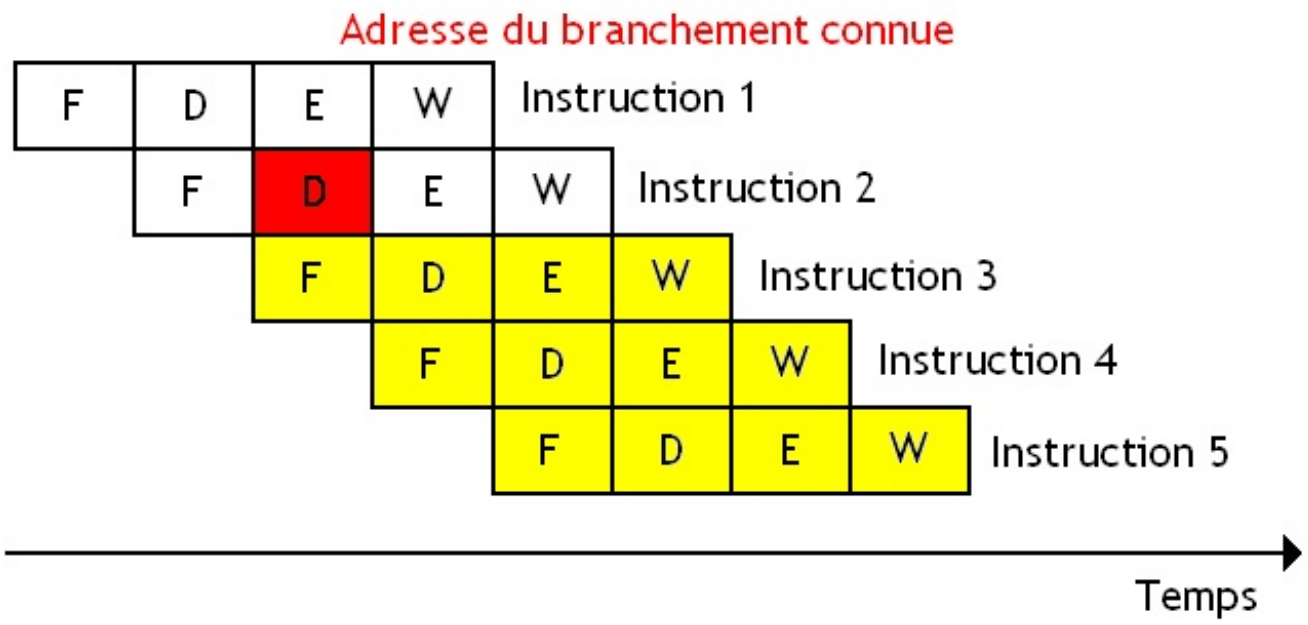
Et comparé à l' History Buffer ?

Cette technique a deux gros avantages comparé à l'*History Buffer*. En cas d'exception, pas besoin de remettre les registres à leur bonne valeur : ces bonnes valeurs sont directement disponibles depuis le *Retiremment Register File*. On supprime ainsi de nombreuses opérations de copie, qui peuvent plomber les performances. Ensuite, on n'a plus besoin d'ajouter un port au *Register File*, qui servait à lire l'ancienne valeur du registre à modifier pour la stocker dans l'*History Buffer*. Et des ports en moins, ça signifie *Register File* plus rapide et qui chauffe moins. Par contre, l'utilisation d'un Future File requiert pas mal de circuits.

Les branchements viennent mettre un peu d'ambiance !

On a vu dans le chapitre sur le pipeline que les branchements avaient un effet plutôt délétère sur celui-ci. Les raisons sont identiques à celles qu'on a vu pour les exceptions. Et c'est normal : les exceptions et interruptions ressemblent fortement aux branchements. Par contre, les techniques utilisées pour gérer les branchement seront différentes, plus adaptées aux branchements.

Lorsqu'on charge un branchement dans le pipeline, l'adresse à laquelle brancher sera connue après un certain temps. Cette adresse de destination est connue au minimum une fois l'instruction décodée, et parfois plus loin. Elle peut être connue durant l'étape de *Decode*, durant l'étape de *Register Read*, ou dans une autre étape : tout dépend du processeur. Et notre branchement va prendre un certain nombre de cycles d'horloge pour arriver à ces étages : l'adresse à laquelle brancher n'est pas connue tout de suite.



Tout se passe comme si notre branchement été retardé, décalé de quelques cycles d'horloge. Durant ce temps de retard, les instructions continueront d'être chargées dans notre pipeline. Il s'agit des instructions en jaune. Et ces instructions sont donc en cours d'exécution dans notre pipeline...alors qu'elles ne devraient pas. En effet, ces instructions sont placées après le branchement dans l'ordre du programme. Logiquement, elles n'auraient pas dues être exécutées, vu que notre branchement est censée avoir fait brancher notre processeur autre part.

Pour éviter tout problème, on doit faire en sorte que ces instructions ne fassent rien de mal, et qu'elles ne modifient pas les registres du processeurs ou qu'elles ne lancent pas d'écritures en mémoire.

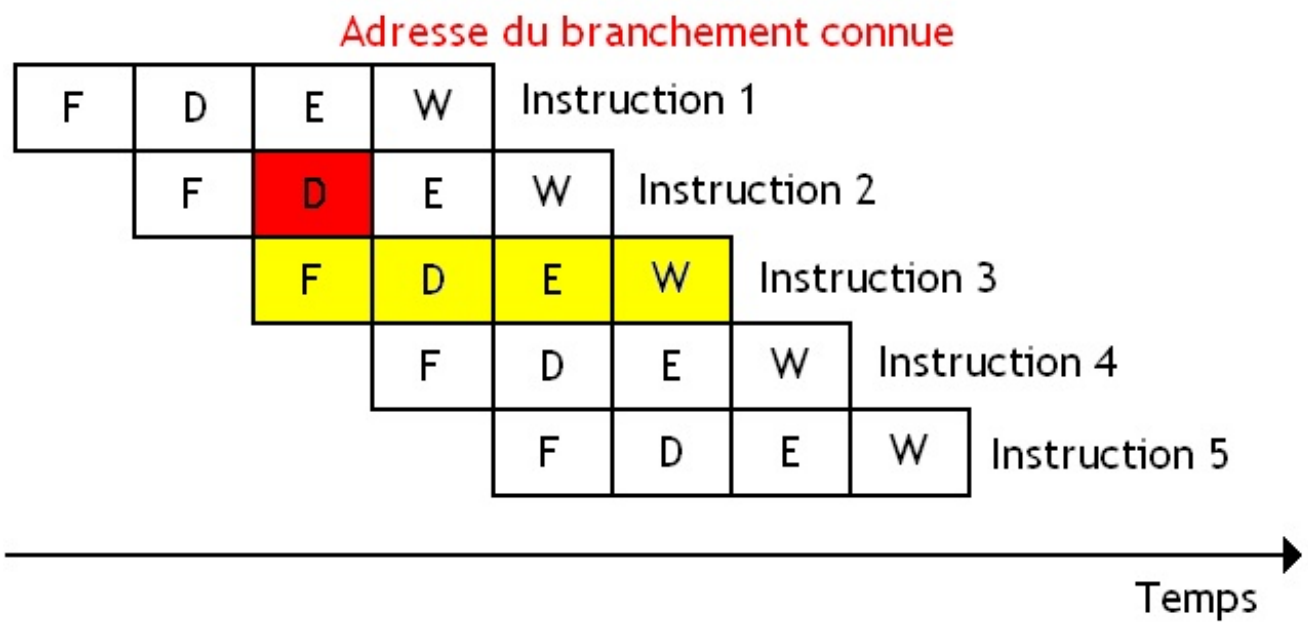
Solutions non-spéculatives

Comme on vient de le voir, nos branchements posent quelques problèmes, et il a bien fallu trouver une solution. Pour cela, quelques solutions simplistes ont été trouvées, et c'est celles-ci dont on va parler en premier lieu.

Délai de branchements

La solution la plus simple consiste à inclure des instructions qui ne font rien à la suite du branchement : c'est ce qu'on appelle un **délai de branchement**. Avec ces instructions, le processeur chargera des instructions qui ne feront rien dans notre pipeline, jusqu'à ce que l'adresse à laquelle brancher soit connue.

Prenons l'exemple d'un pipeline à 4 étages, dans lequel l'adresse à laquelle brancher est connue dans le deuxième étage. Il suffit donc de placer une instruction inutile juste après le branchement. Celle-ci est indiquée en jaune dans le schéma qui suit.



En rajoutant juste ce qu'il faut d'instructions qui ne font rien, on évite les problèmes. On peut limiter la casse en remplissant le vide d'instructions qui suit le branchement par des instructions indépendantes du branchement, mais cela a ses limites : ça n'est pas toujours possible.

Mais pour rajouter juste ce qu'il faut d'instructions, il faut que notre compilateur sache à quel étage du pipeline l'adresse du branchement est connue. Et cela varie suivant les processeurs, même pour des processeurs ayant un jeu d'instruction identique : par exemple, cet étage n'est pas le même entre un Core 2 Duo et un Pentium 4 ! Certains connaissent cette adresse tôt, d'autres plus tard. Autant vous dire qu'avec cette solution, il faudrait recompiler ou reprogrammer tous nos programmes pour chaque processeur.

Pour limiter la casse, on peut faire en sorte que l'adresse à laquelle brancher soit connue le plus tôt possible. En concevant correctement le pipeline, cela peut se faire plus ou moins difficilement. Mais d'autres solutions ont étés inventées, et on verra celles-ci dans deux chapitres.

Branch Free Code

Une première solution pour éviter les problèmes avec les branchements est tout simplement de les remplacer par d'autres instructions assez simples qui ne poseront pas de problème à l'exécution.

Certains calculs souvent utilisés dans un programme peuvent ainsi utiliser des branchements inutilement. Par exemple, le calcul de la valeur absolue d'un nombre, ou le calcul du maximum de deux nombres. Généralement, ces calculs sont effectuées via des instructions de branchements. Vu que ces calculs sont assez communs, le processeur peut fournir des instructions capables d'effectuer ces calculs directement : le jeu d'instruction du processeur peut ainsi contenir des instruction ABS, MIN, MAX, etc. On peut ainsi se passer de branchements pour ces calculs.

Si jamais le processeur ne fourni pas ces instructions, on peut quand même se passer de branchements pour effectuer ces calculs. Il arrive parfois qu'on puisse remplacer certains morceaux de code utilisant des branchements par des morceaux de programmes utilisant des instruction arithmétiques et logiques, sans branchements. Divers *hacks* plus ou moins efficaces (et surtout plus ou moins affreux) existent : allez voir sur google et renseignez-vous sur les différentes techniques de *bit twiddling* ou de *bit hacking*, vous verrez que de nombreux calculs simples peuvent se passer de branchements. Bien sûr, ces techniques ont aussi d'autres qualités : plus on supprime de branchements, plus le compilateur sera apte à modifier l'ordre des instructions pour gérer le pipeline au mieux, et plus il pourra utiliser les registres à bon escient. Utiliser ces technique peut donc être utile, au cas où. Du moins, si vous n'avez pas peur de nuire à la lisibilité de votre programme.

Instructions à prédicats

Ceci dit, fournir des instructions pour supprimer des branchements est une bonne idée qui va plus loin que ce qu'on a vu plus haut. Certains concepteurs de processeurs ont ainsi voulu créer des instructions plus générale, capables de supprimer un plus grand nombre de branchements. Ainsi sont nées les **instructions à prédicat** ! Ces instructions à prédicat sont des instructions "normales", comme des additions, copie d'un registre dans un autre, multiplication, accès mémoire, etc ; avec une différence : elles ne font quelque chose que si une condition est respectée, valide. Dans le cas contraire, celles-ci se comportent comme un `nop`, c'est à dire une instruction qui ne fait rien !

Utilité

Leur but ? Rendre les branchements inutiles pour la construction de petites structures de contrôle de type *Si...Alors* ou *Si...Alors...Sinon*. Avec ces instructions, il suffit d'utiliser une instruction de test et de placer les instructions à exécuter ou ne pas exécuter (en fonction du résultat de l'instruction de test) immédiatement à la suite. Si la condition testée par l'instruction de test est réalisée, nos instructions feront ce qui leur est demandé tandis qu'elles ne feront rien dans le cas contraire : c'est bien ce qu'on attend d'un *Si...Alors* ou d'un *Si...Alors...Sinon*

Défauts

Évidemment, ces instructions ont quelques défauts : elles ne sont utiles que pour des *Si...Alors* ou des *Si...Alors...Sinon* contenant peu d'instructions. Si ceux-ci contiennent beaucoup d'instructions, nos instructions à prédicat ne feront rien durant un moment alors qu'un branchement aurait permis de zapper directement les instructions à ne pas exécuter. De même, ces instructions entrainent l'apparition de dépendances de données : les instructions qui les suivent seront dépendantes de leur résultat. Ce ne serait pas le cas avec d'autres techniques comme la prédiction de branchement. Autant dire que ces instructions ne résolvent pas grand chose et que d'autres techniques doivent impérativement être trouvées.

L'exemple du processeur Itanium d'Intel

Pour donner un exemple d'instructions à prédicats, je vais vous parler des instructions de l'Itanium, un processeur inventé par Intel dans les années 2000. Ce processeur avait une architectures et un jeu d'instruction assez révolutionnaire, avec de nombreuses fonctionnalités innovantes, mais cela n'a pas empêché celui-ci de faire un bide.

L'Itanium ne possède pas de registre d'état. A la place, l'Itanium possède plusieurs registres d'états de un bit ! Il y en a en tout 64, qui sont numérotés de 0 à 63. Chacun de ces registres peut stocker une valeur : vrai (un) ou faux (zéro). Le registre 0 est en lecture seule : il contient toujours la valeur vrai, sans qu'on puisse le modifier. Ces registres sont modifiés par des instructions de comparaison, qui peuvent placer leur résultat dans n'importe quel registre à prédicat. Elle doivent tout de même préciser le registre dans lequel stocker le résultat.

C'est assez différent de ce qu'on trouve sur les processeurs x86. Sur un processeur x86, il n'y a pas de registre à prédicat. A la place, les instructions à prédicat précisent implicitement un bit dans le registre d'état. Une instruction à prédicat est ainsi conçue pour ne lire qu'un seul bit du registre d'état en particulier, et ne peut lire les autres. Mais revenons à l'Itanium.

Chaque instruction à prédicat va préciser quel est le registre qui contient la valeur vrai ou faux permettant d'autoriser ou d'interdire son exécution en utilisant un mode d'adressage spécial. L'instruction s'exécutera normalement si ce registre contient la valeur vrai, et elle ne fera rien sinon. Petite remarque : une instruction peut carrément spécifier plusieurs registres. Ainsi, une instruction peut s'exécuter si deux registres à prédicats sont à vrais. Par exemple, elle peut faire un ET logique sur ces deux bits et décide de s'exécuter si jamais le résultat est true. Elle peut aussi faire un OU logique, un XOR, un NAND, etc.

Les processeurs malins

D'autres processeurs décident de prendre le problème à bras le corps. Ils arrivent à détecter les branchements dans le tout premier étage du pipeline, et arrêtent de charger les instructions tant que l'adresse vers laquelle brancher n'est pas connue. Comme quoi, les Pipelines Bubbles sont vraiment partout !

Conclusion

Ces techniques seraient suffisantes si les branchements n'étaient pas nombreux. Mais ce n'est pas le cas : diverses analyses faites dans les années 1985 ont montrées que dans un programme, environ une instruction sur 15 était un branchement dont plus de la moitié étaient des appels ou retours de sous-programme. Et depuis, la situation est devenue encore pire avec la prééminence de la programmation objet : celle-ci est souvent à l'origine d'un plus grand nombre d'appels de fonctions (dans ce genre de langages, les sous-programmes sont plus courts mais plus nombreux, avec moins d'instructions) et plus de branchements indirects.

Autant dire que les branchements posent un sérieux problème et empêchent d'obtenir de bonnes performances avec notre pipeline. Pour résoudre ce problème, il a fallu trouver des techniques encore plus efficaces pour limiter la catastrophe. Voyons lesquelles !

Prédiction de branchement

Pour éviter ces temps d'attente, les concepteurs de processeurs ont inventé ce qu'on appelle l'**exécution spéculative de branchement**. Cela consiste à essayer de deviner l'adresse vers laquelle le branchement va nous envoyer suivant sa nature, et l'exécuter avant que celui soit décodé et que l'adresse de destination soit connue. On va donc devoir essayer de prédire quel sera le résultat du branchement et vers quelle adresse celui-ci va nous envoyer.

Ce nécessite de résoudre deux problèmes :

- savoir si un branchement sera exécuté ou non : c'est la **prédiction de branchement** ;
- dans le cas où un branchement serait exécuté, il faut aussi savoir quelle est l'adresse de destination : c'est la **prédiction de direction de branchement**.

Pour résoudre le premier problème, notre processeur contient un circuit qui va déterminer si notre branchement sera **pris** (on devra brancher vers l'adresse de destination) ou **non-pris** (on poursuit l'exécution de notre programme immédiatement après le branchement) : c'est **l'unité de prédiction de branchement**. La prédiction de direction de branchement fait face à un autre problème : il faut déterminer l'adresse de destination de notre branchement. Cette prédiction de direction de branchement est déléguée à un circuit spécialisé : **l'unité de prédiction de direction de branchement**. Ces deux unités sont reliées à l'unité en charge du *fetch*, et travaillent de concert avec elle. Dans certains processeurs, les deux unités sont regroupées dans le même circuit.

Erreurs de prédiction

Bien évidemment, une erreur est toujours possible : le processeur peut se tromper en faisant ses prédictions : il peut se tromper d'adresse de destination par exemple. Et le processeur doit éviter tout problème.
On a alors deux cas :

- soit les instructions chargées en avance sont les bonnes : le processeur a donc gagné du temps et continue simplement son exécution ;
- soit les instructions préchargées ne sont pas les bonnes : c'est une **erreur de prédiction**.

Que faire lors d'une erreur de prédiction ?

Vider le pipeline

Dans ce cas, notre pipeline aura commencé à exécuter des instructions qui n'auraient pas du être exécutées. Il faut donc faire en sorte que ces instructions n'enregistrent pas leurs résultats et soient stoppées si possible. La gestion des mauvaises prédiction dépend fortement du processeur, et certains sont plus rapides que d'autres quand il s'agit de gérer ce genre de cas : certains peuvent reprendre l'exécution du programme immédiatement après avoir détecté la mauvaise prédiction (ils sont capables de supprimer les instructions qui n'auraient pas du être exécutées), tandis que d'autres ne peuvent le faire immédiatement et doivent attendre quelques cycles.

Dans le pire des cas, ces instructions qui n'auraient pas du être chargées ne peuvent pas être interrompues et on doit attendre qu'elles aient fini leur exécution : elles vont poursuivre leur chemin dans le pipeline, et progresser étage par étage jusqu'à leur dernière étape. On est donc obligé d'attendre que celle-ci arrivent à leur toute dernière étape avant de pouvoir faire quoique ce soit : il faut que ces instructions quittent le pipeline et que celui-ci soit vidé de ces instructions poubelle. Tant que ces instructions ne sont pas sorties du pipeline, on est obligé d'attendre sans pouvoir charger d'instructions dans notre pipeline. Ce temps d'attente est du temps de perdu.

Exemple : un pipeline à 5 étages :

- **Fe** : Fetching ;
- **De** : Décodage ;
- **Of** : Operand Fetching ;
- **Ex** : Exécution ;
- **Wb** : Write back.

On suppose que l'adresse de destination est connue à l'étape Ex.

On suppose que l'on exécute un branchement JMP dans notre pipeline et que celui-ci est mal prédit : le processeur se trompe d'adresse de destination. Ce qui fait que tant que notre adresse destination réelle n'est pas connue, notre processeur continue à charger les instructions à partir de la fausse adresse de destination qu'il a prédit.

Dans le tableau qui va suivre, X représente des instructions qui doivent être exécutées normalement et Ix représente des instructions qui n'auraient pas du être execuétes. Les tirets représentent des étages qui ne font rien.

état du pipeline	Fe	De	Of	Ex	Wb
Normal	Jmp	X1	X2	X3	X4
Normal	I1	Jmp	X1	X2	X3
Vidage du pipeline	I2	I1	Jmp	X1	X2
Vidage du pipeline	-	I2	I1	Jmp	X1
Vidage du pipeline	-	-	I2	I1	Jmp
Vidage du pipeline	-	-	-	I2	I1
Vidage du pipeline	-	-	-	-	I2
Restauration du pipeline	Jmp	X1	X2	X3	X4
Normal	X265	Jmp	X1	X2	X3

A la ligne 3, le processeur repère une erreur de prédiction : au cycle d'horloge suivant, il ne chargera pas d'instruction et commencera à vider le pipeline. Une fois la pipeline vidé, il faudra restaurer celui-ci à son état originel (avant l'exécution du branchement) pour éviter les problèmes, et recommencer à exécuter les bonnes instructions..

On remarque une chose importante dans notre exemple : cela prend 5 cycles pour vider le pipeline, et notre pipeline possède 5 étages. Ce temps d'attente nécessaire pour vider le pipeline est égal au nombres d'étages de notre pipeline. Sachez que c'est vrai tout le temps, et pas seulement pour des pipelines de 5 étages. En effet, la dernière instruction à être chargée dans le pipeline le sera durant l'étape à laquelle on détecte l'erreur de prédiction : il faudra attendre que cette instruction quitte le pipeline, et qu'elle passe par tous les étages de celui-ci.

Restaurer le pipeline

De plus, il faut remettre le pipeline dans l'état qu'il avait avant le chargement du branchement dans le pipeline. Tout se passe lors de la dernière étape. Et pour cela, on réutilise les techniques vues dans le chapitres précédent. Les techniques qui nous servaient pour les exceptions sont réutilisées pour les mauvaises prédictions de branchements. Dans tous les cas, notre processeur doit détecter automatiquement les erreurs de prédiction : ça utilise pas mal de circuits assez complexes.

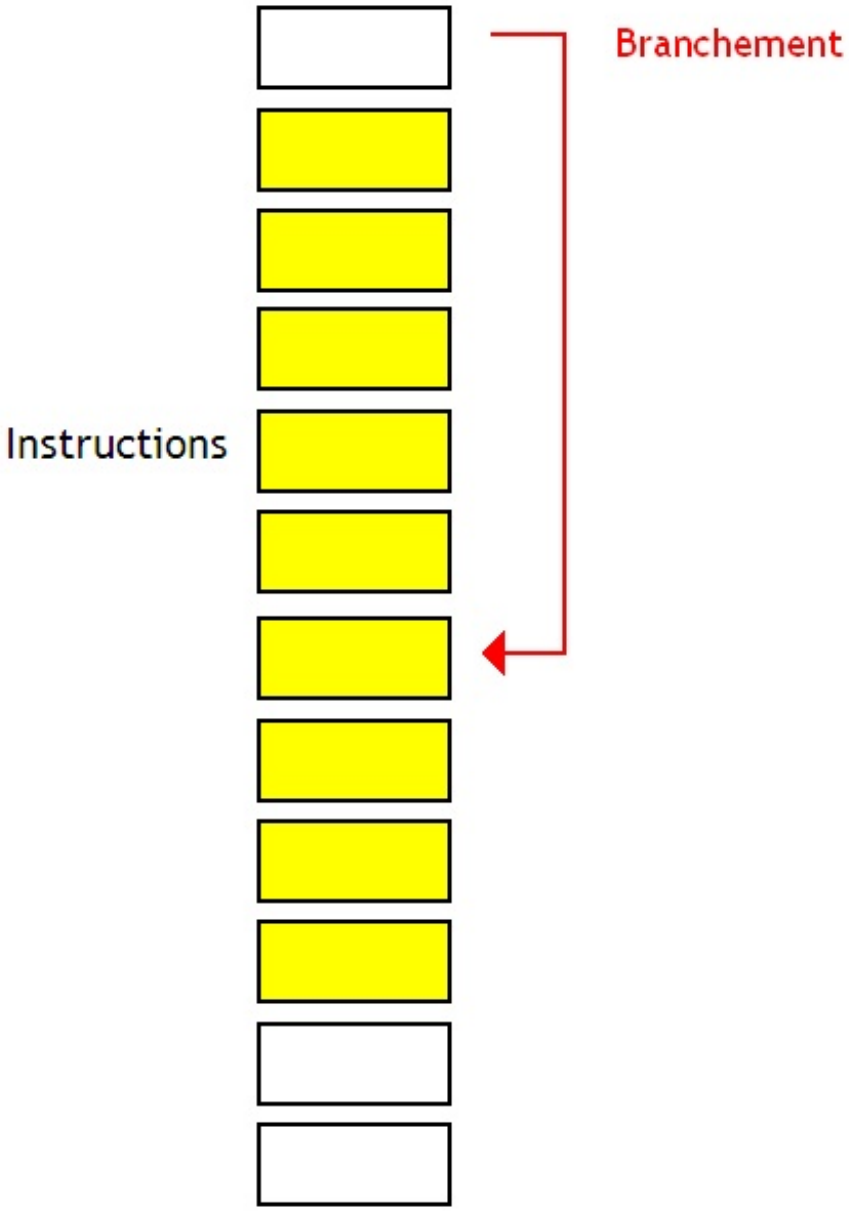
Première solution : faire en sorte que les modifications et calculs effectuées par les instructions fautives ne soient pas enregistrées en mémoire ou dans les registres. Il faut donc faire en sorte que la dernière étape de ces instructions ne fasse rien et n'enregistre pas les résultats de ces instructions. Le résultat de l'instruction, est recopié dans le registre final lors de leur dernière étape si aucune erreur de prédiction n'a eu lieu. Dans le cas contraire, les modifications et manipulations effectuées par notre instruction ne sont pas écrites dans le registre ou la case mémoire voulue et le tout est simplement oublié et perdu. Pour cela, on peut forcer l'ordre de l'enregistrement des résultats en mémoire, afin que ceux-ci se fassent dans l'ordre, et autoriser les écritures à la fin du pipeline. On peut aussi laisser nos instructions enregistrer leurs résultats dans le désordre : il suffit de stocker les résultats de nos instructions une sorte de mémoire tampon qu'on appelle le *Reorder Buffer*.

Deuxième solution : on laisse nos instructions écrire leurs résultats et manipuler leurs résultats et on restaure les données valides, qui auront préalablement étés sauvegardées quelque part dans le processeur. Cela peut se faire soit en utilisant un *History Buffer*, ou un *Future File*.

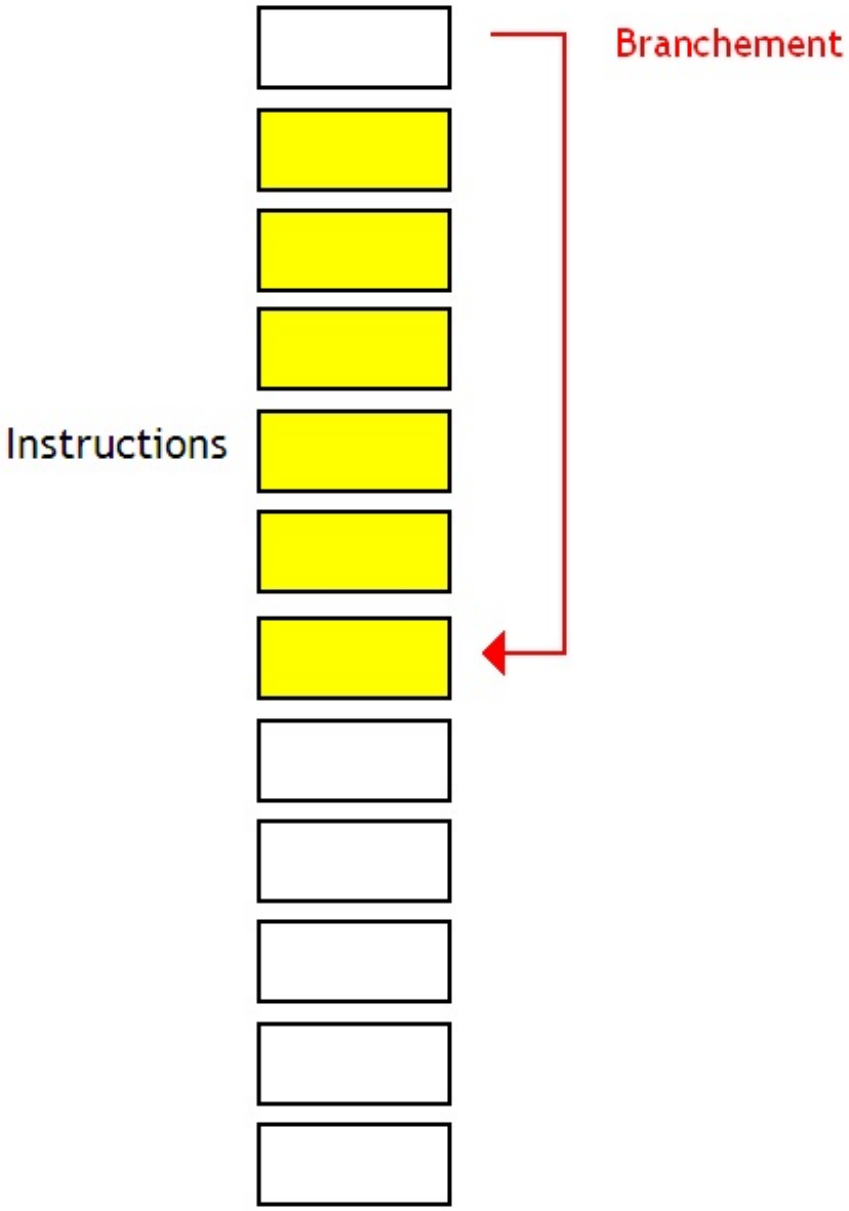
Minimal Control Dependancy

Avec les techniques vues plus haut, toutes les instructions qui suivent un branchement dans notre pipeline sont supprimées. Leur résultat ne sera pas enregistré dans la mémoire ou dans les registres, ou alors toutes les modifications qu'elles feront seront annulées. Mais pourtant, certaines d'entre elles pourraient être utiles.

Prenons un exemple : supposons que l'on dispose d'un processeur de 31 étages (un Pentium 4 par exemple). Supposons que l'adresse du branchement est connue au 9éme étage. On fait face à un branchement qui envoie le processeur seulement 6 instructions plus loin. Si toutes les instructions qui suivent le branchement sont supprimées, on obtient ceci : les instructions en jaune sont celles qui auraont étès supprimées lors de la vidange du pipeline.



On remarque pourtant que certaines instruction sont potentiellement correctes : celles qui suivent le point d'arrivée du branchement. Elles ne le sont pas forcément : il se peut qu'elles aient des dépendances avec les instructions supprimées. Mais si elles n'en ont pas, alors ces instructions auraient du être exécutées. Il serait donc plus efficace de les laisser enregistrer leurs résultats au lieu de les ré-exécuter à l'identique un peu plus tard.



Ce genre de choses est possible sur les processeurs qui implémentent une technique du nom de Minimal Control Dependancy. En gros, cette technique fait en sorte que seules les instructions qui dépendent du résultat du branchement soient supprimées du pipeline en cas de mauvaise prédiction.

Prédiction de direction de branchement

Quand un branchement est non-pris, on sait exactement quelles instructions charger : celles qui suivent en mémoire. Il n'y a alors pas de difficulté.

Mais dans le cas où on considère un branchement pris, que faire ? A quelle adresse le branchement va-t-il nous envoyer ?

Sans la réponse à cette question, impossible de charger les bonnes instructions, et on doit alors attendre que l'adresse de destination soie connue : en clair, effectuer un délai de branchement.

La prédiction de direction de branchement sert à compenser ce manque en prédisant quelle sera l'adresse de destination. Bien sur, une unité de prédiction de direction de branchement n'est pas un circuit extralucide, mais utilise quelques algorithmes simples pour déduire l'adresse de destination.

Cette prédiction est plus ou moins facile suivant le type de branchements. La prédiction de branchement se moque qu'un branchement soit conditionnel ou inconditionnel, ce qui l'intéresse c'est de savoir si le branchement est :

- un **branchement direct**, pour lequel l'adresse de destination est toujours la même ;
- ou un **branchement indirect**, pour lequel l'adresse de destination est une variable et peut donc changer durant exécution du programme.

Les branchements directs sont plus facilement prévisibles : l'adresse vers laquelle il faut brancher est toujours la même. Pour les branchements indirects, vu que cette adresse change, la prédire celle-i est particulièrement compliqué (quand c'est possible).

Voyons maintenant comment notre unité de prédiction de direction de branchement va faire pour prédire quelle sera l'adresse de destination de notre branchement.

Branch Barget Buffer

Une solution à ce problème a été trouvée dans certains cas : lorsqu'un branchement direct est exécuté plusieurs fois dans un programme dans un court intervalle de temps. L'idée consiste à se souvenir de l'adresse de destination du branchement lors de sa première exécution. Si notre branchement est un branchement direct, cette adresse de destination sera toujours la même (sauf cas particuliers tellement anecdotiques qu'on ferait bien de ne pas en parler) : ainsi, lors des exécutions suivante du même branchement, on saura d'avance quelle est l'adresse de destination.

Pour se souvenir de l'adresse de destination d'un branchement, on a besoin d'une petite "mémoire" capable de retenir pour chaque branchement, l'adresse vers laquelle celui-ci nous fait brancher : le **Branch Target Buffer**.

Ce branchement est une instruction qui est placée à une adresse bien précise. Notre branchement est ainsi identifié dans le *Branch Target Buffer* par son adresse en mémoire. Celui stocke l'adresse du branchement, et l'adresse de destination.

Ne pas confondre les deux adresses !

Ce *Branch Target Buffer* est souvent implémenté comme un cache *fully associative* : il est découpé en lignes de cache qui contiennent l'adresse de destination du branchement et dont le *tag* contient l'adresse du branchement. Pour prédire l'adresse de destination d'un branchement, il suffit de vérifier les tags de chaque ligne et de comparer avec l'adresse du branchement à exécuter : si un tag correspond, on a alors un *Branch Target Buffer Hit* et le *Branch Target Buffer* contient l'adresse de destination du branchement. Dans le cas contraire, on ne peut pas prédire l'adresse de destination du branchement : c'est un *Branch Target Buffer Miss*.

Le fonctionnement de l'algorithme de prédiction de branchement basé sur un *Branch Target Buffer* est simple. A la première exécution du branchement, on attend donc que le branchement s'exécute et on mémorise l'adresse du branchement ainsi que l'adresse de destination. A chaque nouvelle exécution de ce branchement, il suffit de lire l'adresse de destination contenue dans le *Branch Target Buffer* et continuer l'exécution du programme à cette adresse. Il va de soit que cette technique ne peut rien lors de la première exécution d'un branchement.

De plus, cela ne marche pas dans certains cas impliquant des branchements indirects, pour lesquels l'adresse de destination peut varier. Lorsque cette adresse de destination change, le *Branch Target Buffer* ne sert à rien et une erreur de prédiction a lieu.

Pour information, sachez que notre *Branch Target Buffer* ne peut stocker qu'un nombre limité d'adresses. Sur les processeurs x86 actuels, le nombre d'adresses est d'environ 64, et varie suivant le processeur. Quand le *Branch Target Buffer* est rempli et qu'un nouveau branchement s'exécute, on supprime les informations d'un branchement du *Branch Target Buffer* pour faire de la place. Cela peut poser un problème : un branchement qui aurait pu être exécuté dans un futur proche peut se retrouver supprimé du *Branch Target Buffer*. On ne peut prédire l'adresse vers laquelle il branchera alors qu'avec un *Branch Target Buffer* plus grand, on aurait pu. Cela s'appelle un *BTB Miss*.

Pour limiter la casse, certains *Branch Target Buffer* ne mémorisent pas les branchements non-pris (du moins, ceux qui n'ont jamais été pris auparavant). Cela permet de faire un peu de place et évite de remplacer des données utiles par une adresse qui ne servira jamais. D'autres optimisations existent pour éviter de remplir le *Branch Target Buffer* avec des adresses mémoires de branchement inutiles, et permettre de garder un maximum de contenu utile dans le *Branch Target Buffer*. Pour cela, quand un branchement doit être stocké dans notre *Branch Target Buffer* et que celui-ci est plein, il faut choisir quel branchements enlever du *Branch Target Buffer* avec le plus grand soin. Si vous vous souvenez, notre *Branch Target Buffer* est une mémoire cache : et bien sachez que les algorithmes vus dans le chapitre sur la mémoire cache fonctionnent à merveille !

Prédiction des branchements indirects

Sur les processeurs qui n'implémentent pas de techniques capables de prédire l'adresse de destination d'un branchement indirect, le processeur considère qu'un branchement indirect se comporte comme un branchement direct : le branchement va brancher vers l'adresse destination utilisée la dernière fois qu'on a exécuté le branchement. Et c'est normal : seule l'adresse de

destination valide lors de la dernière exécution du branchement est stockée dans le *Branch Target Buffer*. A chaque fois que ce branchement change d'adresse de destination, on se retrouve avec une mauvaise prédiction. Tous les processeurs moins récents que le Pentium M prédisent les branchements indirects de cette façon.

Certains processeurs haute performance sont capables de prédire l'adresse de destination d'un branchement indirect. A une condition cependant : que l'adresse destination change de façon répétitive, en suivant une régularité assez simple.

Ces techniques de prédiction de branchement indirect utilisent un *Branch Target Buffer* amélioré. Ce *Branch Target Buffer* amélioré est capable de stocker plusieurs adresses de destination pour un seul branchement. De plus, ce *Branch Target Buffer* amélioré stocke pour chaque branchement et pour chaque adresse de destination des informations qui lui permettent de déduire plus ou moins efficacement quelle adresse de destination est la bonne.

Mais même malgré ces techniques avancées de prédiction, les branchements indirects et appels de sous-programmes indirects sont souvent très mal prédits, même avec un *branch predictor* optimisé pour ce genre de cas. Pour information, ce type de branchement est devenu plus fréquent avec l'apparition des langages orientés objets, les langages procéduraux utilisant peu de branchement indirects (c'est d'ailleurs une des raisons qui font que les langages orientés objets sont plus lents que les langages procéduraux).

Return Fonction Prédictor

Certains processeurs peuvent prévoir l'adresse à laquelle il faudra reprendre lorsqu'un sous-programme a fini de s'exécuter : si vous vous souvenez, notre sous-programme fini par un branchement inconditionnel indirect qui fait reprendre notre programme où il en était avant l'exécution de notre sous-programme. Si vous vous souvenez, cette adresse de retour est stockée sur la pile, ou dans des registres spéciaux du processeurs dans certains cas particuliers.

Certains processeurs possèdent un circuit spécialisé capable de prédire l'adresse de retour d'une fonction : le *Return Fonction Prédictor*. Lorsqu'une fonction est appelée, ce circuit stocke l'adresse de retour d'une fonction dans des "registres" internes au processeur organisés sous forme d'une pile. Avec cette organisation des registres en forme de pile, on sait d'avance que l'adresse de retour du sous-programme en cours d'exécution est au sommet de cette pile. Quand on tombe donc sur une instruction de retour de sous-programme, il suffit de lire l'adresse au sommet de cette pile de registre pour savoir vers où brancher.

Ce nombre de registres est limité : le *Return Fonction Prédictor* ne peut conserver les adresses de retour que d'un nombre limité de branchements. Généralement, la limite tourne autour de 8 (c'est le cas du processeur Atom d'Intel) ou 16 (sur les processeurs les plus performants). Imbriquer trop de sous-programmes l'un dans l'autre peut parfois poser quelques problèmes : certains de ces sous-programmes seront alors mal prédits.

Prédiction de branchement

Maintenant, voyons comment notre processeur fait pour prédire si un branchement est pris ou non.

Prédiction statique

L'idée derrière la prédiction statique est simple. Suivant le branchement, on considère que celui-ci est soit toujours pris, soit jamais pris. L'idée est de faire une distinction entre les différents types de branchements qui ont beaucoup de chance d'être pris et ceux qui ne seront jamais ou presque jamais pris.

Il faut savoir que :

- un branchement inconditionnel est toujours pris, de même qu'un branchement conditionnel dont la condition de saut est vérifiée ;
- et qu'un branchement conditionnel dont la condition de saut n'est pas respectée est non-pris.

Ainsi, si on prédit qu'un branchement est non-pris, on pourra continuer l'exécution des instructions qui suivent le branchement dans la mémoire sans problème. A l'inverse si le branchement est prédit comme étant pris, le processeur devra recourir à l'unité de prédiction de direction de branchement.

L'algorithme de prédiction statique le plus simple est de faire une distinction entre branchements conditionnels et branchements inconditionnels. Par définition un branchement inconditionnel est toujours pris, tandis qu'un branchement conditionnel peut être pris ou ne pas être pris.
Ainsi, on peut donner un premier algorithme de prédiction dynamique :

- les branchement inconditionnels sont toujours pris ;
- les branchements conditionnels ne sont jamais pris.

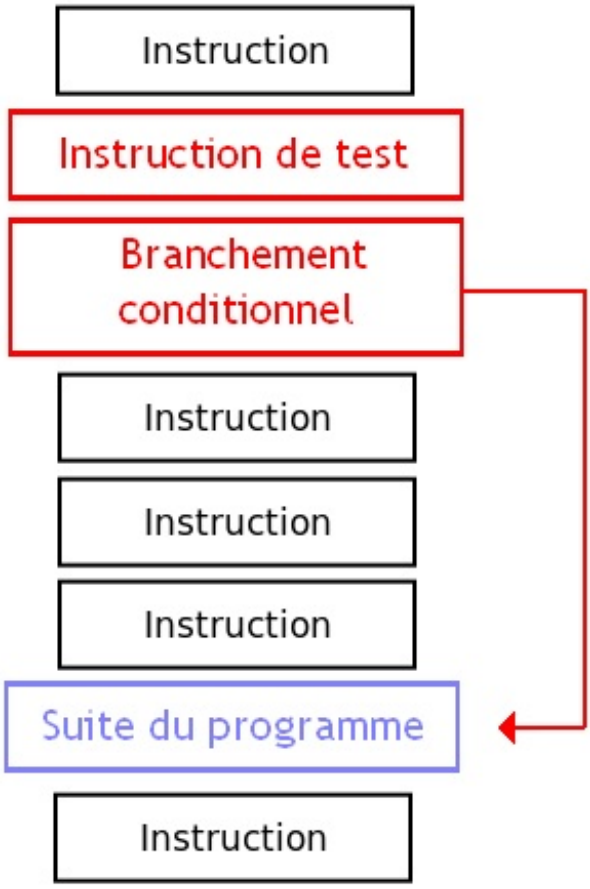
Lors de l'exécution d'un branchement conditionnel, le processeur continue d'exécuter les instructions qui suivent le branchement dans la mémoire programme jusqu'à ce que le résultat du branchement conditionnel soit disponible. Si la prédiction était fausse, on recommençait à partir de l'adresse vers laquelle pointait le branchement et continuait sinon.

Cette méthode est particulièrement inefficace pour les branchements de boucles, où la condition est toujours vraie, sauf en sortie de boucle ! Il a donc fallu raffiner légèrement l'algorithme de prédiction statique.

Une autre manière d'implémenter la prédiction statique de branchement est de faire une différence entre **les branchements conditionnels ascendants** et les **branchements conditionnels ascendants**.

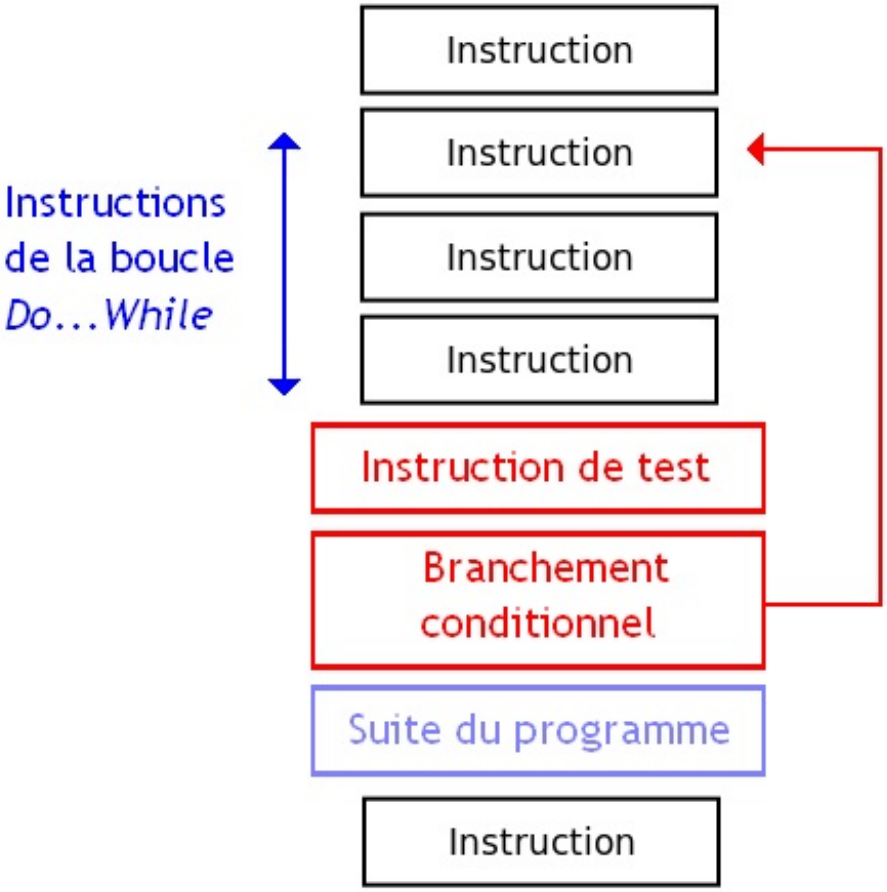
Un branchement conditionnel ascendant, aussi appelé *forward branch*, est un branchement qui demande à notre programme de reprendre à une instruction dont l'adresse est supérieure à l'adresse du branchement. En clair, le branchement demande au processeur de reprendre plus loin dans la mémoire.

Par exemple, prenons une structure de contrôle *Si...Alors* : le branchement conditionnel est un *forward branch*. Ce branchement renvoie à des instruction ayant une adresse plus élevée : l'instruction vers laquelle on branche est placée après le branchement.



Un branchement conditionnel descendent, aussi appelé *backward branch*, est un branchement qui demande à notre programme de reprendre à une instruction dont l'adresse est inférieure à l'adresse du branchement. En clair, le branchement demande au processeur de reprendre plus loin dans la mémoire.

Par exemple, regardons la conception en assembleur d'une boucle *Do...While* : celle-ci est fabriquée avec un *backward branch*. Ce branchement renvoie à des instruction ayant une adresse plus faible, plus haute : l'instruction vers laquelle on branche est placée avant le branchement.



Hors, si vous regardez la traduction en assembleur des structures de contrôle utilisées dans les langages de programmation structurés actuels, vous remarquerez que sous quelques hypothèses (condition d'un if toujours pris et que les boucles exécutent au moins une itération), les *forward branch* sont rarement pris et les *backward branch* sont presque toujours pris. On peut ainsi modifier l'algorithme de prédiction statique comme suit :

- les branchements inconditionnels sont toujours pris ;
- les *backward branch* sont toujours pris ;
- les *forward branch* ne sont jamais pris .

Bien sûr, cette prédiction statique est basée sur quelques hypothèses, comme mentionné plus haut : un *Si...Alors* a sa condition vérifiée et le branchement conditionnel n'est pas pris, par exemple (généralisable au *Si...Alors...Sinon*, avec quelques subtilités). Un programmeur peut ainsi, s'il connait la traduction de ses structures de contrôle en assembleur, tenter de respecter au maximum ces hypothèses afin d'optimiser ses structures de contrôle et leur contenu pour éviter que les branchements de ses structures de contrôle soient mal prédits.

Sur certains processeurs, certains bits de l'opcode d'un branchement peuvent permettre de préciser si notre branchement est majoritairement pris ou non-pris : ces bits spéciaux permettent d'influencer les règles de prédiction statique et de passer outre les réglages par défaut. Ainsi, un programmeur ou un compilateur peut donner à l'unité de prédiction de branchement des informations sur un branchement en lui disant que celui-ci est très souvent pris ou non-pris. Ces informations peuvent faire passer outre les mécanismes de prédiction statique utilisés normalement. On appelle ces bits des ***Branch Hints***.

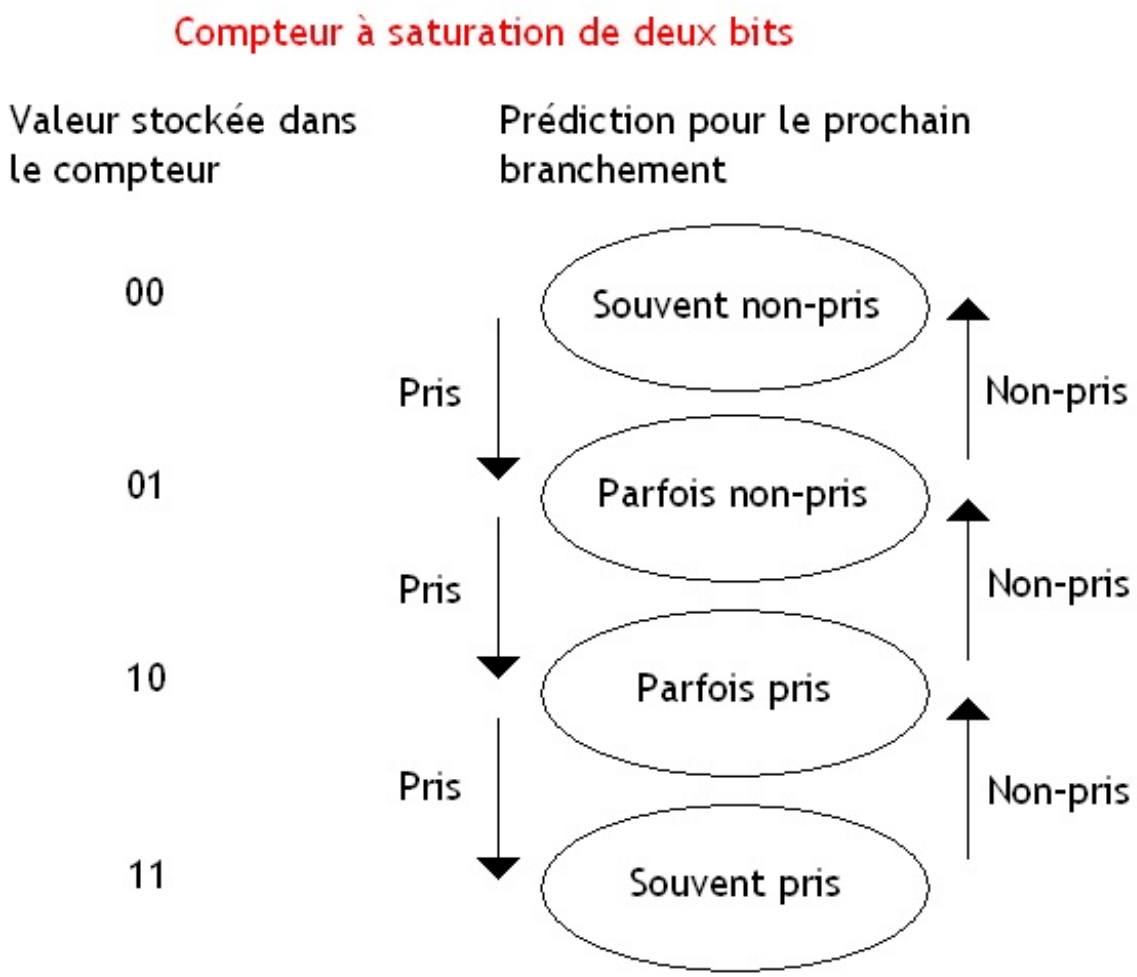
Compteurs à saturation

Avec la prédiction dynamique, il n'y a pas besoin de supposer que le branchement est pris ou pas suivant sa nature : l'unité de prédiction de branchement peut le deviner dans certains cas. L'implémentation la plus simple de la prédiction dynamique est basée sur la technique des compteurs à saturation. Pour cela, **on mémorise à chaque exécution du branchement si celui-ci est pris ou pas, et on effectue une moyenne statistique sur toutes les exécutions précédentes du branchement**.

Par exemple, un branchement qui a déjà été exécute quatre fois, qui a été pris une seule fois et non-pris les 3 autres fois, a une chance sur 4 d'être pris et trois chances sur quatre d'être pris. L'idée est donc de se souvenir du nombre de fois qu'un branchement a été pris ou non-pris pour pouvoir déduire statistiquement la probabilité que ce branchement soit pris. Si cette probabilité est supérieure à 50%, le branchement est considéré comme pris. Dans le cas contraire, notre branchement est considéré comme non-pris.

L'idée, c'est d'utiliser un compteur dédié pour chaque branchement présent dans le *Branch Target Buffer*, qui mémorisera le nombre de fois qu'un branchement est pris ou non-pris. Ces compteurs sont des circuits électroniques qui stockent un nombre entier de **N** bits, dont la valeur est comprise entre 0 et $2^N - 1$. Ce nombre entier peut être augmenté ou diminué de 1 selon le résultat de l'exécution du branchement (pris ou pas).

Lorsqu'un branchement doit être exécuté, l'unité de prédiction de branchement regarde si celui-ci a son adresse dans le *Branch Target Buffer*. Si ce n'est pas le cas, on réserve une partie de cette mémoire pour ce branchement et le compteur est initialisé à une certaine valeur, généralement égale à peu-prêt la moitié de N. Par la suite, si ce branchement est pris, le compteur sera augmenté de 1. Dans le cas contraire, le compteur sera diminué de 1.



Pour décider si le branchement sera pris ou pas, on regarde le bit de poids fort du nombre stocké dans le compteur : si vous regardez le schéma au-dessus, vous remarquerez que le branchement est considéré comme pris si ce bit de poids fort est à 1. Dans le cas contraire, notre branchement est donc considéré comme non-pris.

Certains processeurs ont une unité de branchement qui se limite à un seul compteur à saturation auquel on a jouté un *Branch Target Buffer* et de quoi effectuer de la prédiction statique si jamais le branchement n'a jamais été exécuté ou si celui-ci n'est pas un branchement conditionnel. C'est le cas du processeur Intel Pentium 1, par exemple. Pour la culture générale, il faut savoir que le compteur à saturation du Pentium 1 était légèrement buggé.

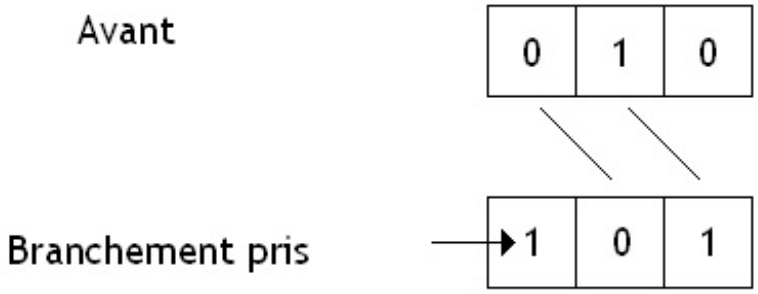
Two level adaptative predictor

Le seul problème avec un compteur à saturation unique, c'est que celui-ci ne marche bien que lorsqu'un seul branchement est exécuté à la fois : ce compteur ne stocke les probabilités d'être pris ou non-pris que pour un seul et unique branchement. En clair, celui-ci ne fonctionne bien que pour les boucles ! Par exemple, un branchement qui est exécuté comme suit : pris, non-pris, pris, non-pris, etc ; sera mal prédit une fois sur deux par notre compteur à saturation. Même chose pour un branchement qui ferait : pris, non-pris, non-pris, non-pris, pris, non-pris, non-pris, non-pris, etc.

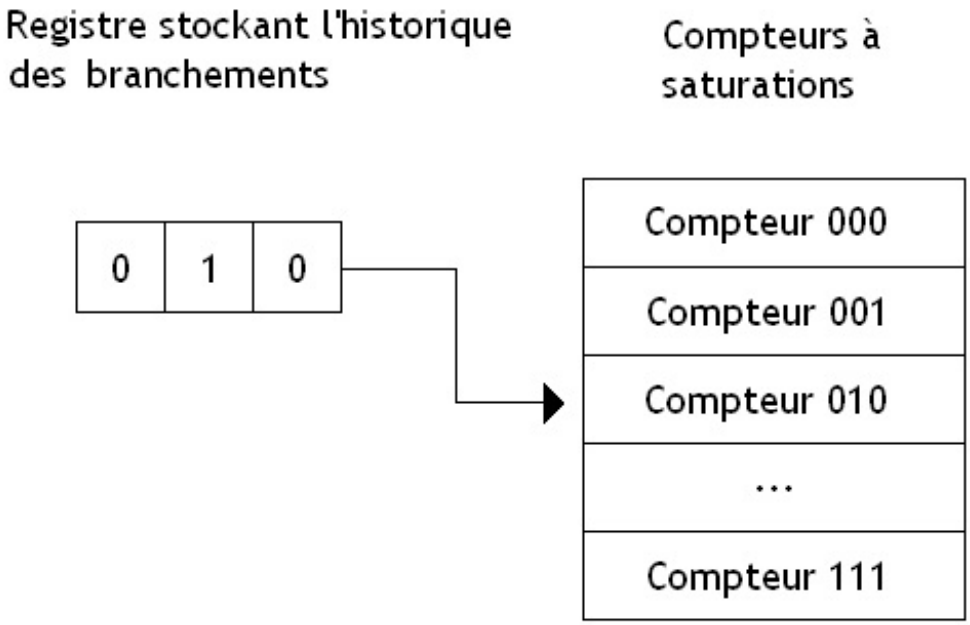
Une solution pour régler ce problème est de se souvenir des 2, 3, 4 (ou plus suivant les modèles) exécutions précédentes du branchement. Ainsi, un branchement qui ferait pris, non-pris, pris, non-pris, etc ; sera parfaitement prédit si l'unité de prédiction

de branchement est capable de se souvenir des deux exécutions précédentes du branchement. Un branchement qui ferait : pris, non-pris, non-pris, non-pris, pris, non-pris, non-pris, non-pris, etc ; demandera une unité de prédiction de branchements capable de se souvenir des 4 dernières exécutions d'un branchement.

Pour cela, on va utiliser un registre qui stockera l'historique des derniers branchements exécutés. Ce registre est ce qu'on appelle un **registre à décalage**. Ce registre à décalage fonctionne exactement comme un registre qu'on aurait couplé à un décaleur (ici, un décaleur par 1), qui est chargé de décaler le contenu de notre registre. A chaque fois qu'un branchement s'exécute, on décale le contenu du registre et on fait rentrer dans celui-ci un 1 si le branchement est pris, et un zéro sinon.



Pour chaque valeur possible contenue dans le registre, on trouvera un compteur à saturation qui permettra de prédire quelle est la probabilité que le prochain branchement soit pris. Ce registre est donc couplé à plusieurs compteurs à saturations : pour un registre de n bits (qui se souvient donc des n derniers branchements exécutés), on aura besoin de 2^n compteurs à saturation.



Chacun de ces compteurs permettra mémoriser le nombre de fois qu'un branchement a été pris à chaque fois que que celui-ci a été exécuté après s'être retrouvé dans une situation telle que décrite par le registre. Par exemple, si le registre contient 010, le compteur associé à cette valeur (qui est donc numéroté 010), sert à dire : à chaque fois que je me suis retrouvé dans une situation telle que le branchement a été non-pris, puis pris, puis non-pris, le branchement a été majoritairement pris ou non-pris.

En utilisant une unité de prédiction de branchement de ce type, on peut prédire tout branchement qui suivrait un schéma qui se répète tous les n branchements ou moins, en utilisant un registre d'historique de n bits (et les 2^n compteurs à saturation qui vont avec).

Certains processeurs se payent le luxe d'avoir un registre (et le jeu de compteurs à saturation qui vont avec) différent pour chaque branchement qui est placé dans le *Branch Target Buffer* : on dit qu'ils font de la **prédiction locale**.

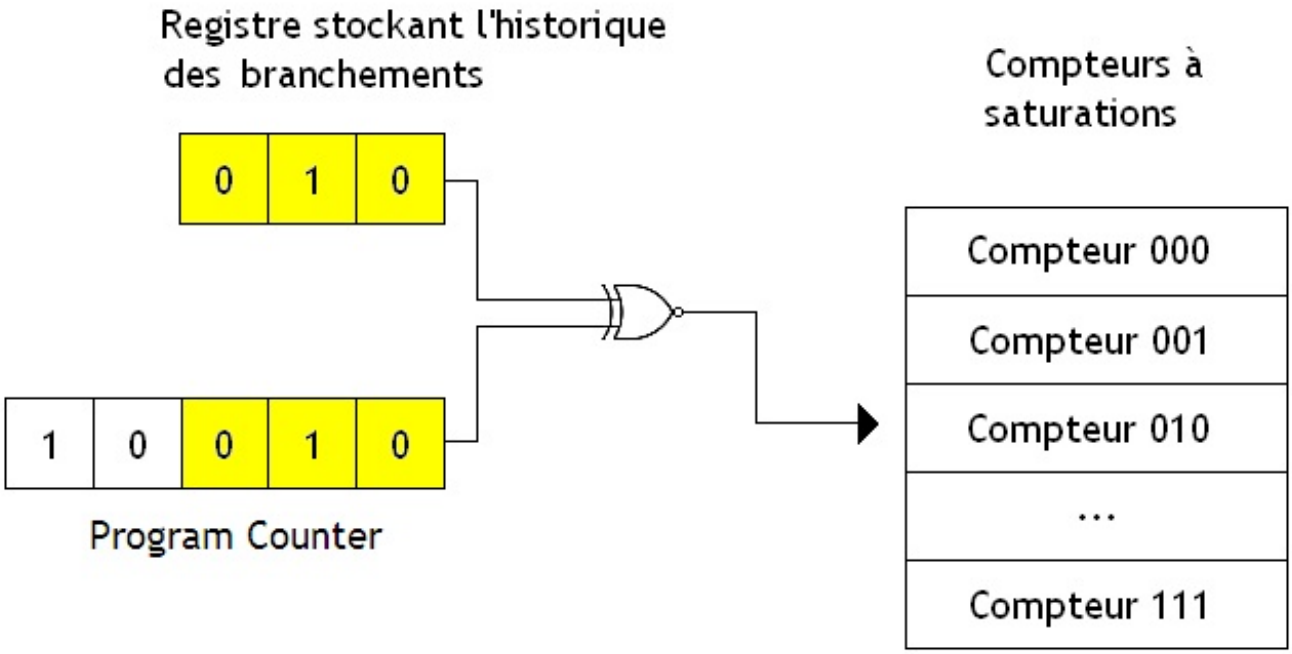
Global Predictor

D'autres, préfèrent utiliser un registre unique pour tous les branchements : ceux-ci font de la font de la **prédiction globale**. Ces unités de prédiction de branchement incorporent un seul registre d'historique, qui est utilisé pour tous les branchements. C'est par exemple le cas du Pentium 4E qui utilise une unité de branchement capable de se souvenir des 16 derniers branchements précédents. Les processeurs AMD 64 utilisent quand à eux un *two level adaptative predictor* avec un registre d'historique de 8 bits.

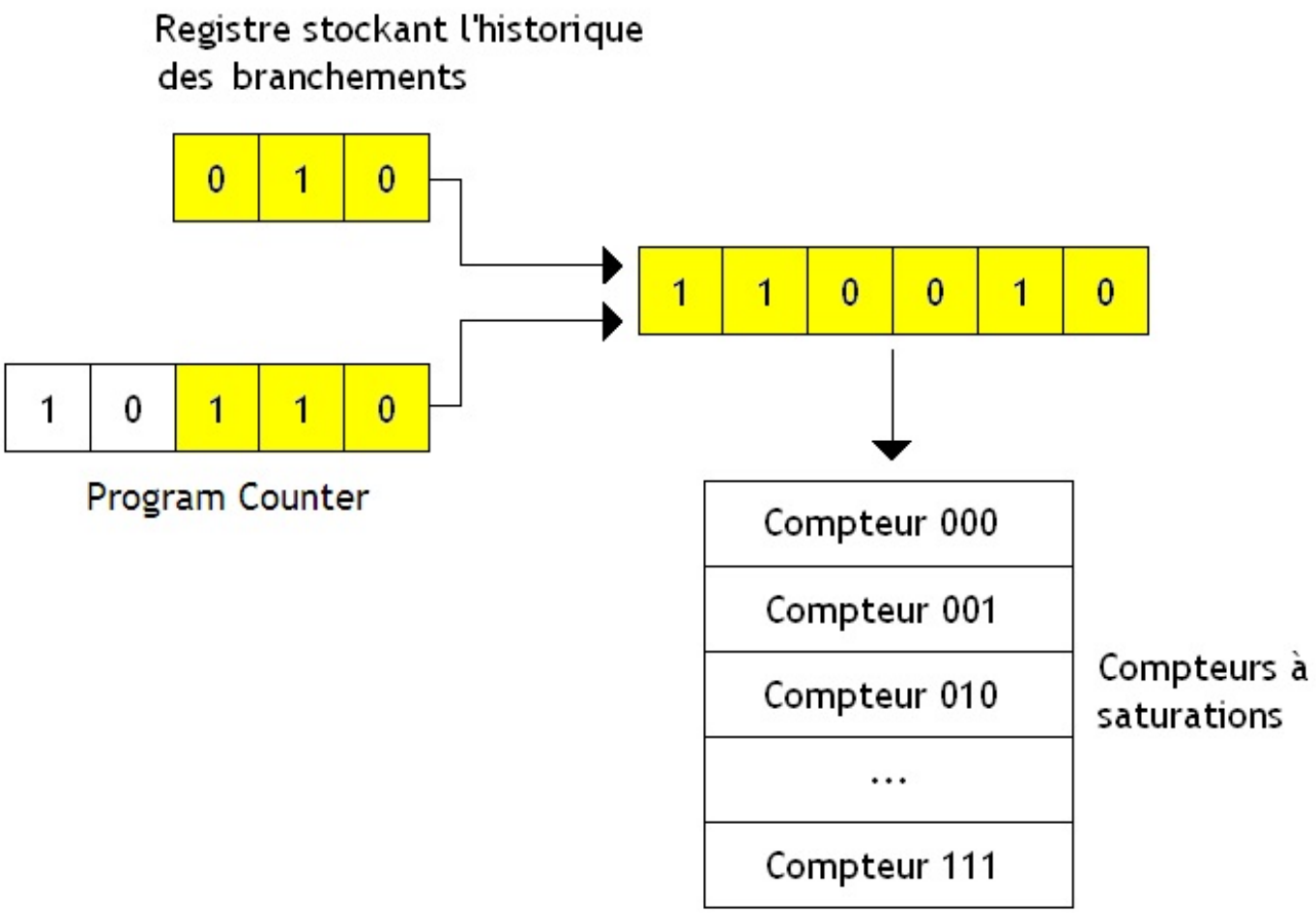
L'avantage de ces unités de prédiction, c'est que d'éventuelles corrélations entre branchements sont prises en compte. Mais cela a un revers : si deux branchements différents passent dans l'unité de prédiction de branchement avec le même historiques, alors ils modifieront le même compteur à saturation. En clair : ces deux branchements vont se marcher dessus sans vergogne : chacun des branchements va interférer sur les prédiction de l'autre!

Bien évidemment, il faut absolument trouver une solution. Et c'est la raison d'être des unités de prédiction Gshare et Gselect. Avec ces unités de prédiction, limiter le plus possible les cas dans lesquels deux branchements différents avec la même historique utilisent le même compteur à saturation. Pour cela, on va effectuer une petite opération entre l'historique et certains bits de l'adresse de ces branchements (leur Program Counter) pour trouver quel compteur utiliser.

Avec les prédicteurs Gshare, cette opération est un simple XOR entre le registre d'historique et les bits de poids faible de l'adresse du branchement. Le résultat de ce XOR donne ne numéro du compteur à utiliser.



Avec les prédicteurs Gselect, cette opération consiste simplement à mettre cote à cote ces bits et l'historique pour obtenir le numéro du compteur à saturation à utiliser.

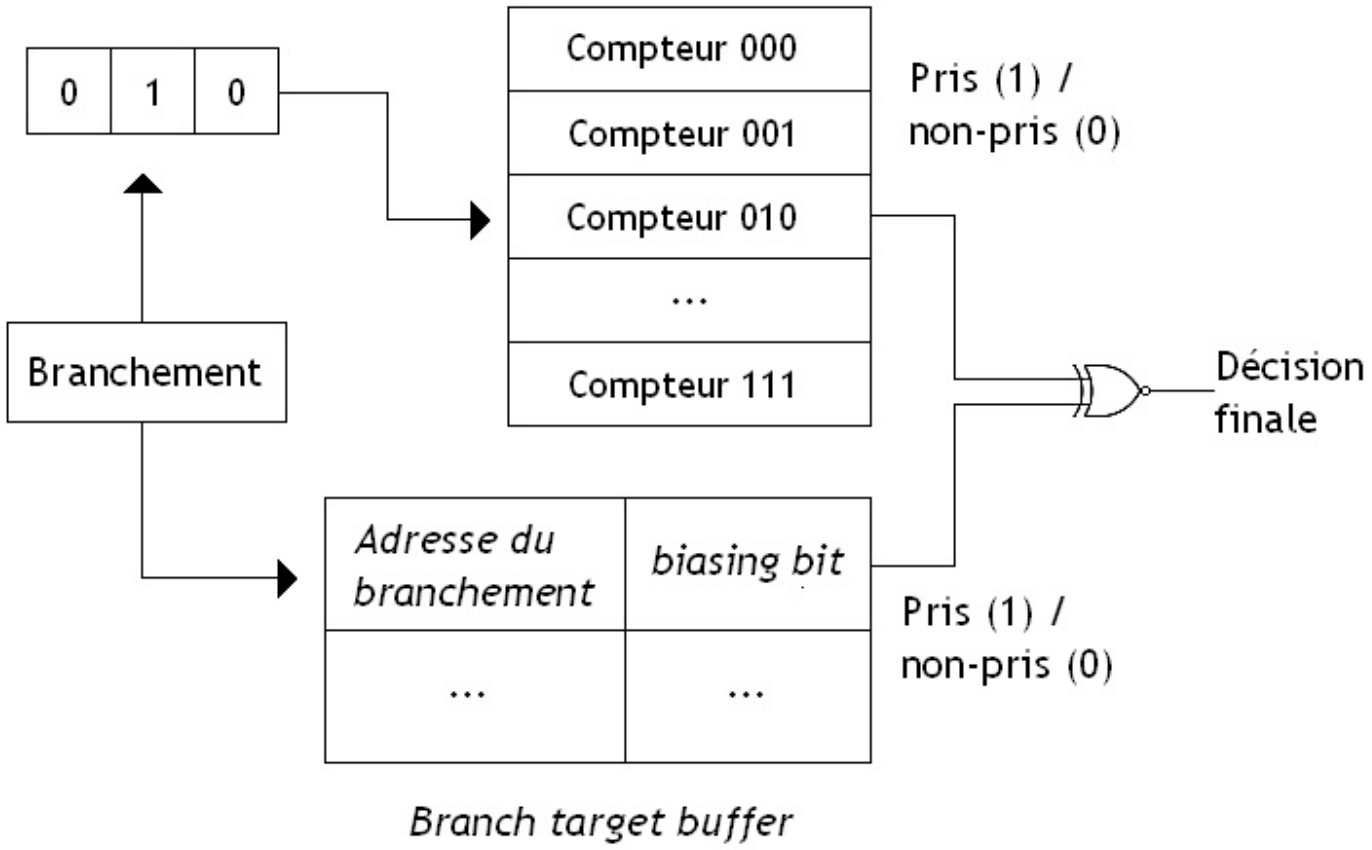


Agree Predictor

Le *Two level adaptative predictor* possède toutefois un léger problème : avec celui-ci, on prédit qu'un branchement est pris ou non-pris en fonction des branchements qui le précédent et qui ont chacun mit à jour les compteurs à saturation dans notre *Two level adaptative predictor*. Ces compteurs à saturation ne sont pas réservés à un branchement et sont partagés : seul le contenu du registre d'historique permet de sélectionner un de ces compteurs à saturation.

Et le problème vient de là : deux branchements différents, exécutés à des moments différents peuvent devoir modifier le même compteur à saturation : il suffit que les branchements qui précédent ces branchements aient positionnés le registre d'historique à la même valeur. Et dans ce cas là, un branchement sera prédit non pas en fonction de son historique (est-ce que celui-ci a déjà étè pris ou non-pris beaucoup de fois), mais aussi en fonction des exécutions de branchements qui lui sont indépendants et qui ont modifiés le même compteur. Et ces interférences ont tendance à entrainer l'apparition de mauvaises prédictions qui auraient pus être évitées. Il faut donc trouver un moyen de limiter la casse.

Pour cela, 4 chercheurs ont décidés d'inventer une nouvelle unité de prédiction de branchements capable de résoudre de problème : ***l'agree predictor***. Cet *agree predictor* va permettre de se souvenir pour chaque branchement individuel si celui-ci est plus souvent pris ou non-pris en stockant cette information dans un bit spécial nommé le *biasing bit*. Ainsi, notre *agree predictor* est divisé en deux circuits : un *Two level adaptative predictor* normal, et un circuit qui va calculer le *biasing bit*. Ce *biasing bit* est calculé avec l'aide d'un compteur à saturation, relié au *branch target buffer*. Chaque branchement sera identifié par son adresse dans le *Branch target buffer* (ce qui permet d'individualiser la prédiction en fonction du branchement pour éviter les interférences) et chaque branchement aura un compteur à saturation rien que pour lui, qui permettra de savoir si notre branchement est souvent pris ou pas.



Le décision prendra compte des deux circuits de prédiction de branchement en vérifiant que ces deux unités de branchement sont d'accord. Lorsque les deux unités de branchement ne sont pas d'accord, c'est qu'il y a surement eu interférence. Cette décision se fait donc en utilisant une porte ***XOR*** suivie d'une porte ***NON*** entre les sorties de ces deux unités de prédictions de branchement : faites la table de vérité pour vous en convaincre !

Loop Prédictor

Maintenant, il faut savoir que les unités de prédiction de branchement des processeurs haute performance modernes sont capables de prédire à la perfection les branchements de certaines boucles : les boucles *FOR*, ainsi que quelques autres boucles devant s'exécuter N fois (N pouvant varier à l'exécution). De telles unités de prédiction de branchement sont mêmes capables de prédire correctement le branchement qui fait sortir de la boucle en le considérant comme pris !

Comment ces unités de prédiction de branchement sont capables d'une telle prouesse ?

C'est très simple : leur unité de branchement contient un circuit spécialisé dans la prédiction des branchements de ce genre de boucles.

Pour rappel, pour une boucle devant s'exécuter N fois (N dépendant de la boucle), les branchements permettant d'implémenter cette boucle vont devoir s'exécuter N fois : ils vont être pris $N-1$ fois, le dernier branchement étant mal prédit. Donc, la solution est d'utiliser un compteur qui compte de de un en un à partir de zéro. A chaque fois que le branchement est exécuté, on augmente la valeur contenue dans le compteur de 1. Tant que la valeur stockée dans le compteur est différente du nombre N, on considère que le branchement est pris. Si le contenu de ce compteur vaut N, le branchement n'est pas pris.

Mais comment savoir la valeur de ce nombre N ?

Lorsqu'une boucle est exécutée la première fois, ce nombre N est stocké quelque part dans le *Branch Target Buffer*. Cette technique a donc une limite : il faut que la boucle soit exécutée plusieurs fois sans que les informations du branchements ne soient écrasées par d'autres dans le *Branch Target Buffer*. La première fois, les branchements de la boucle sont prédits en

utilisant la prédiction statique ou les compteurs à saturation ou toute autre méthode de prédiction de branchement. De plus, N doit rester la même à chaque exécution de la boucle.

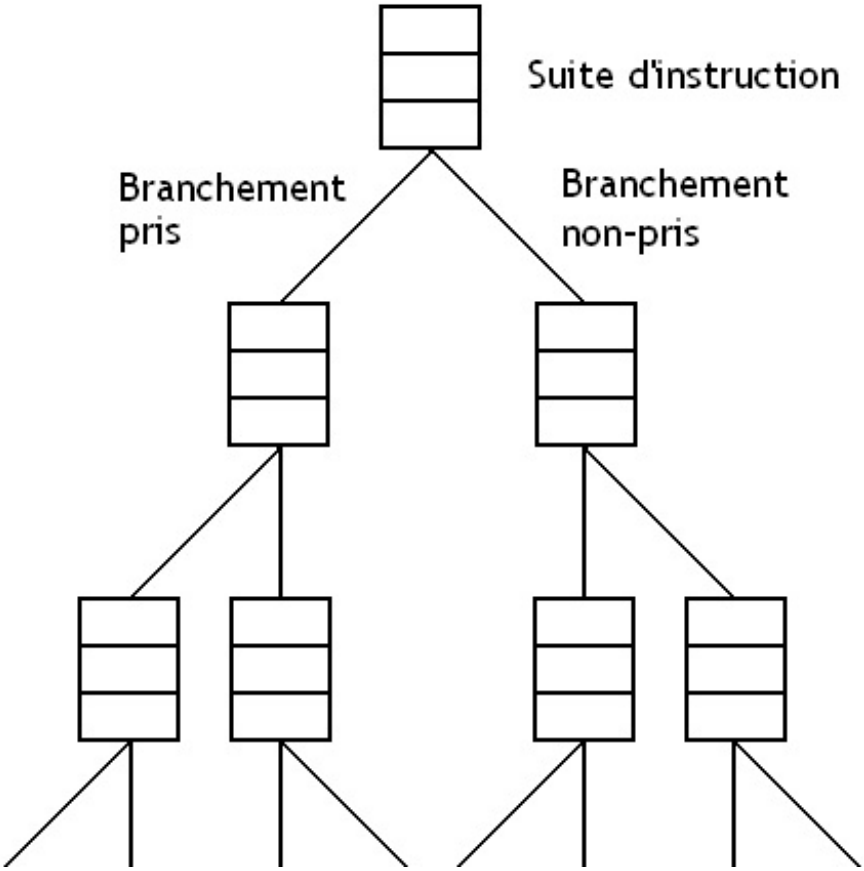
De plus, le *Loop Prédictor*, qui ne peut prédire correctement qu'un nombre limité d'exécution d'un branchement, égal au nombre maximal de valeurs différentes stockables dans son compteur. Exemple : si un branchement utilisé pour concevoir une boucle doit s'exécuter plus que 64 fois (au hasard 5000 fois), et que le *Loop Predictor* utilise un compteur pouvant compter jusqu'à 64 (valeur la plus grande connue à ce jour, valable pour un core 2 duo), alors on aura une mauvaise prédiction tout les 64 exécution du branchement. Autre détail : les branchements à l'intérieur des boucles interfèrent avec le fonctionnement du *Loop Predictor* et empêchent celui-ci de voir qu'il a affaire à une boucle : une autre méthode de prédiction des branchements sera utilisée.

Eager execution

Comme vous le savez, un branchement peut être pris ou non-pris. Le principe de la prédiction de branchement est de choisir une de ces deux possibilités et de commencer à l'exécuter en espérant que ce soit la bonne. Les unités de prédiction de branchement se sont donc améliorées de façon à éviter de se tromper, et choisissant la possibilité la plus probable. Mais si on réfléchit bien, il y a une autre manière de faire : on peut aussi exécuter les deux possibilités séparément, et choisir la bonne une fois qu'on connait l'adresse de destination du branchement. C'est ce qu'on appelle l'***Eager Execution***.

Pour information, sachez qu'il s'agit d'une technique assez avant-gardiste, qui n'est pas encore présente dans les processeurs de nos ordinateurs. Vous pouvez considérer que ce tutoriel est à la pointe de la technologie.

Bien sûr, on est pas limité à un seul branchement, mais on peut poursuivre un peu plus loin.



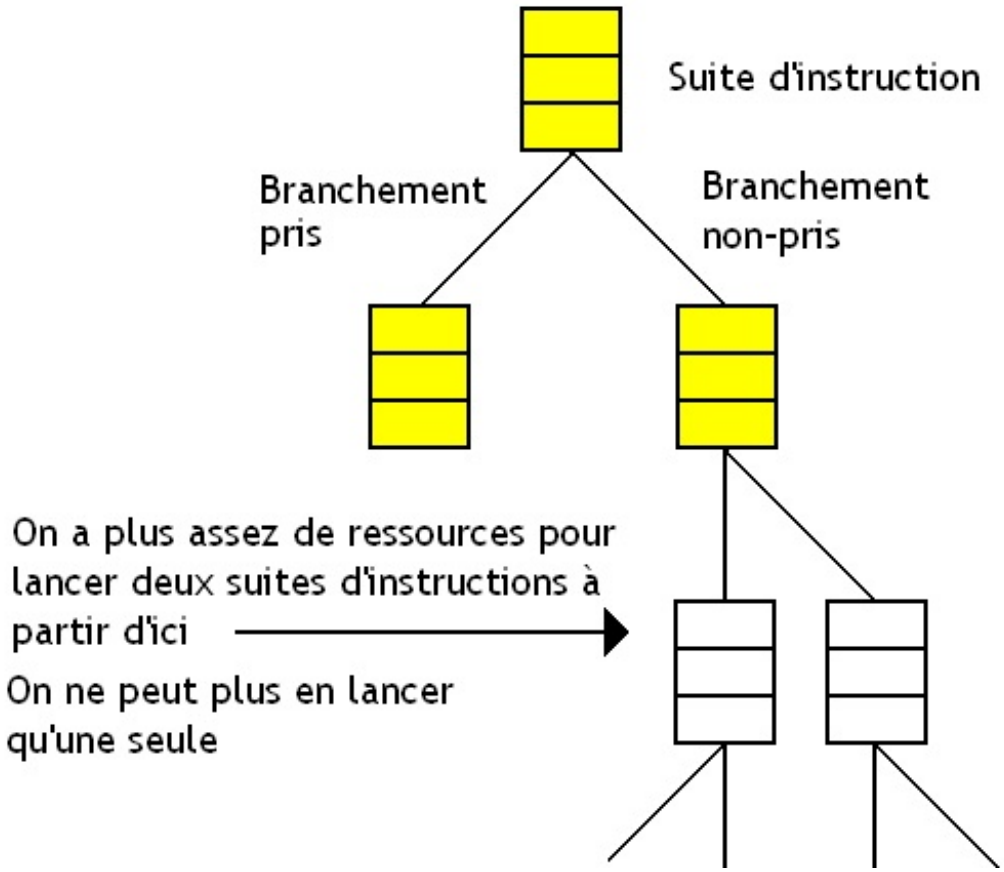
On peut remarquer que cette technique ne peut pas se passer totalement de la prédiction de branchement. Pour pouvoir charger les deux suites d'instructions correspondant à un branchement pris et non-pris, il nous faut forcément savoir quelle est la destination du branchement. On a donc encore besoin de notre unité de prédiction de direction de branchement. Mais on peut théoriquement se passer de l'unité de prédiction de branchement, chargée de savoir si le branchement est pris ou non-pris. J'ai bien dit théoriquement...

Quelques limites pratiques

Le seul problème, c'est qu'on fini rapidement par être limité par le nombre d'unités de calculs dans le processeur, le nombre de registres, etc. L'*eager execution* est donc une bonne solution, mais elle ne peut suffire à elle seule.

Prenons cet exemple : on a déjà rencontré un branchement. On en rencontre un deuxième, mais on ne peut pas lancer les deux

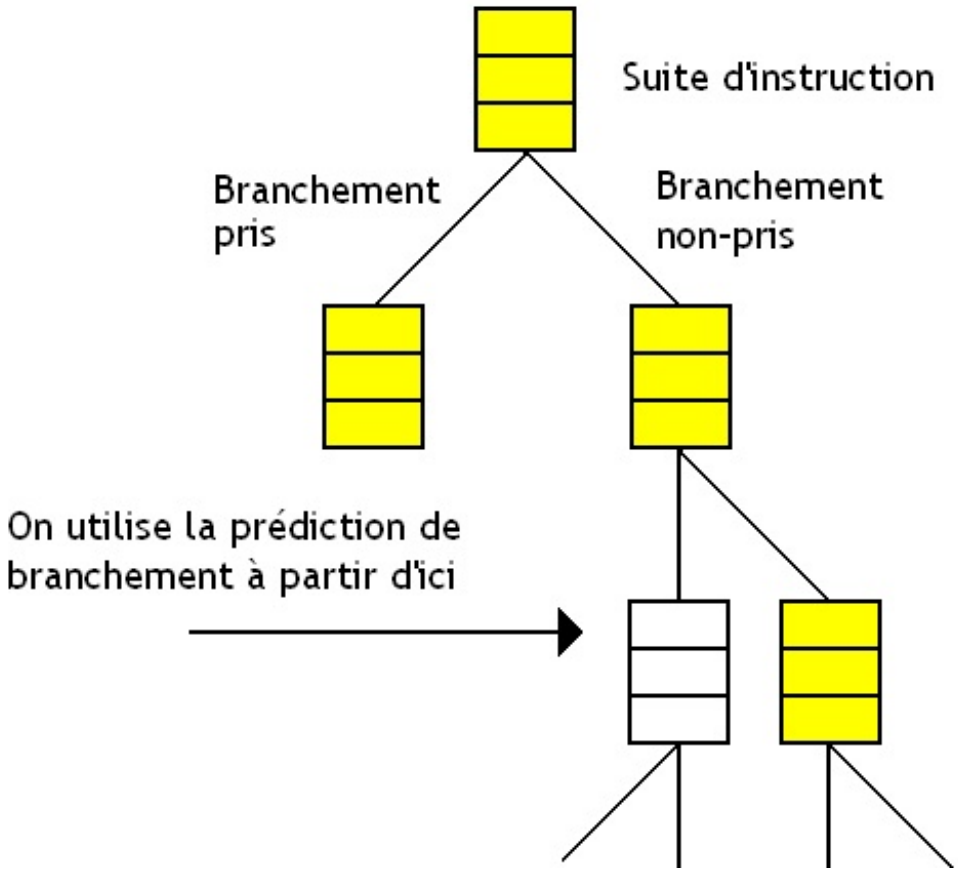
suites d'instructions correspondant aux cas pris et non-pris. On ne peut plus en lancer qu'une.



Que faire ?

La solution est très simple : on peut décider de n'exécuter qu'un seul des cas (pris ou non-pris, histoire d'utiliser notre processeur au mieux. Et oui : on peut coupler l'*eager execution* avec des unités de prédictions de branchement.

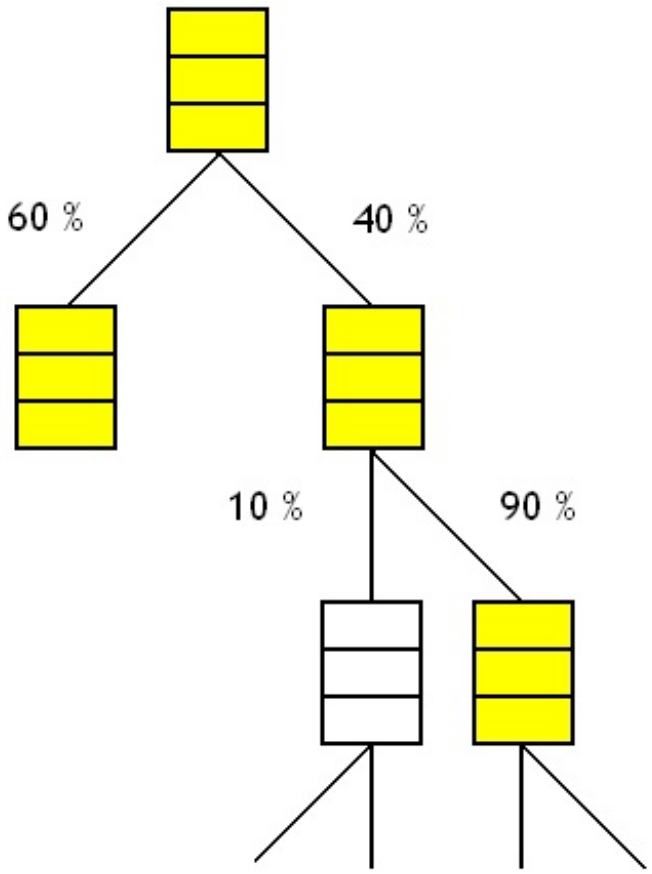
Généralement, on préfère commencer par éxecuter les premiers branchements rencontrés via *eager execution*, et si trop de branchements sont rencontrés alors qu'on a beaucoup de branchements encore en attente (on ne sait pas encore vers où ils branchent), alors on prédit les branchements suivants via la prédiction de branchements. Cela permet d'éviter de trop utiliser d'unités de calculs ou de registres en exécutant deux suites d'instructions.



Disjoint Eager Execution

D'autres techniques existent, et quitte à parler d'*eager execution*, autant parler de la plus efficace : la ***Disjoint Eager Execution*** . C'est la technique qui est censée donner les meilleurs résultats d'un point de vue théorique.

Avec celle-ci, on n'exécute pas les suites d'instructions correspondant à un branchement pris ou non-pris si leur probabilité est trop faible. Cela permet d'économiser un peu plus de ressources . On peut ainsi demander à une unité de prédiction de branchement de calculer la probabilité qu'un branchement soit pris ou non-pris. Ensuite, le processeur n'exécute pas les branchements qui ont les probabilités d'exécution les plus faibles.



A chaque fois qu'un nouveau branchement est rencontré, le processeur refait les calculs de probabilité. Cela signifie que d'anciens branchements qui n'avaient pas étés exécutés car ils avaient une probabilité trop faible peuvent être exécuté si des branchements avec des probabilités encore plus faibles sont rencontrés en cours de route.
Il existe encore d'autres techniques de prédiction de branchements, qu'on a pas encore vu ici : par exemple, on n'a pas abordé l'***alloyed predictor***. Et l'on a pas abordé les diverses optimisations possibles sur les circuits vus précédemment. Pour vous en donner une idée, sur certains processeurs, on utilise carrément deux unités de prédiction de branchements : une très rapide mais pouvant prédire les branchements "simples à prédire" , et une autre plus lente qui prend le relais quand la première unité de prédiction de branchement n'arrive pas à prédire un branchement. La prédiction est ainsi plus rapide pour certains branchements.

Il existe aussi divers projets de recherche, plus ou moins bien avancés, qui cherchent des moyens de prédire encore plus efficacement nos branchements. Certains pensent ainsi utiliser des unités de prédiction de branchement basées sur des réseaux de neurones : le résultat serait assez lent, mais très efficace sur certains branchements. Enfin, certains cherchent à adapter les différentes méthodes de prédiction de branchement sur les instructions à prédicat, afin de pouvoir rendre celle-ci plus efficaces. La recherche est encore en cours, et on peut s'attendre à quelques gains dans les années qui suivent, même si ces gains seront surement à relativiser. Les marges de manœuvre sont assez faibles, et les branchements resteront un problèmes durant un moment encore.

Conclusion

Pour tout ceux qui veulent savoir comment nos branchements sont gérés par les processeurs actuels, téléchargez donc le troisième fichier PF présent sur ce site : http://www.agner.org/optimize/. A l'intérieur ce celui-ci vous aurez de nombreuses informations sur le fonctionnement des processeurs "récents", avec notamment une revue des différentes techniques de prédiction de branchements utilisées.

Dépendances de données

Dans le chapitre précédent, j'ai parlé des dépendances d'instructions et j'ai dit qu'il s'agissait de situations dans lesquelles une instruction avait besoin du résultat d'une ou de plusieurs instruction précédentes pour s'exécuter. J'ai aussi parlé des dépendances structurelles, qui ont tendance à mettre un peu d'ambiance. Ces dépendances sont un des points faibles des processeurs possédant un pipeline. Et pour que nos pipelines deviennent vraiment intéressants, il a bien fallu trouver des solutions pour supprimer l'effet de ces dépendances. Dans ce chapitre, on verra que l'imagination des ingénieurs et des chercheurs a permit de trouver de très belles solutions à ce genre de problèmes. Nous verrons les techniques de *bypass*, d'exécution *out-of-order*, et le *register renamming*, ainsi que quelques autres technologies de pointes que les processeurs modernes implémentent dans leurs circuits. Mais tout d'abord, parlons un peu plus précisément de ces fameuses dépendances : il faut bien commencer par les bases !

Dépendances d'instructions

Une...dépendance ? Nos instructions sont droguées ?

Mais non !

Bien que le terme soit un peu bizarre, il a une signification simple : deux instructions ont une dépendance quand elles cherchent à manipuler la même ressource. Il se peut que deux instructions en cours d'exécution dans le pipeline aient besoin d'accéder au même registre, à la même unité de calcul ou à la même adresse mémoire en même temps. Une des deux instructions devra alors attendre que la ressource voulue soit libre, en patientant dans un étage du pipeline.

Il existe divers types de dépendances, qui ont toutes des origines ou des propriétés différentes. Ces dépendances sont au nombre de trois : dépendances de contrôle, dépendances structurelles, et dépendances de données. Les dépendances de contrôle sont simplement celles dues au branchements et aux exceptions/interruptions : ce qui est après un branchement dépend du résultat du branchement. On a vu tout cela en long en large et en travers dans les deux chapitres précédents. Il ne nous reste plus qu'à voir les dépendances structurelles et les dépendances de contrôle.

Dépendances structurelles

Il se peut qu'un circuit du processeur doive être manipulé par plusieurs instructions à la fois : deux instructions peuvent ainsi vouloir utiliser l'unité de calcul en même temps, ou accéder à la mémoire simultanément, etc. Dans ce genre de cas, il est difficile à un circuit de traiter deux instructions à la fois s'il n'est pas conçu pour. Le processeur fait alors face à une **dépendance structurelle**.

Mais quelles peuvent être ces dépendances structurelles ?

Pour vous montrer à quoi peut ressembler une dépendance structurelle, rien de mieux que de parcourir le pipeline et de lister quelles sont les dépendances qui peuvent survenir.

Exemple

Prenons un exemple : on va charger une instruction devant aller chercher une donnée dans la mémoire dans notre pipeline de 7 étages vu au-dessus. Voici le résultat :

T	T+1	T+2	T+3	T+4	T+5	T+6
PC	Fetch	Decode	Read Operand	Exec	MEM	Writeback
	PC	Fetch	Decode	Read Operand	Exec	MEM
		PC	Fetch	Decode	Read Operand	Exec
			PC	Fetch	Decode	Read Operand
				PC	Fetch	Decode
					PC	Fetch
						PC

Regardez ce qui se passe sur les deux colonnes de la fin : on doit effectuer une étape de *Fetch*, et l'étape de *MEM* simultanément. Or, ces deux étapes doivent aller effectuer des lectures en mémoire. On est en plein aléas structurel : notre mémoire ne peut logiquement effectuer qu'un seul accès à la fois.

Solution

Pour éviter tout problème à cause de ces dépendances, on peut placer des instructions inutiles dans notre programme, afin de décaler nos instructions dans le pipeline. Par exemple, si deux instructions sont besoin d'utiliser le même circuit dans un cycle d'horloge, on peut décider d'en retarder une d'un cycle d'horloge. Il suffit juste d'insérer des instructions qui ne font rien au bon endroit dans notre programme. Mais ce n'est pas une solution très élégante : on perd pas mal en performance en faisant exécuter des instructions "inutiles".

Sinon, on peut laisser la gestion de ces dépendances au processeur. Sur toutes les instructions qui bataillent pour accéder au circuit, le processeur peut donner l'accès au circuit à une instruction à la fois. A chaque cycle, le processeur choisira une instruction et les autres attendrons leur tour bien sagement, bloquées dans un étage du pipeline.

Mais ce genre d'aléas se règle souvent en **dupliquant la ressource à partager**. Il suffit de dupliquer le circuit à partager : au lieu de partager un circuit, on en réserve d'avance un par instruction, et c'est réglé ! C'est pour cela que le cache L1 est coupé en deux caches (un pour les instructions et un autre pour les données). L'unité de *Fetch* peut ainsi charger une (voire plusieurs) instruction depuis le cache L1 sans être gênée par l'étage *MEM* : chacun de ces étages pourra accéder à un cache séparé (L1 instruction pour l'unité de *Fetch*, et L1 donnée pour l'étage de *MEM*). Dupliquer une unité de calcul, des registres, ou tout autre circuit est quelque chose d'assez commun pour éviter d'avoir à partager une ressource, et éviter ainsi les aléas structurels.

Mais cela a tout de même un gros défaut : cela bouffe beaucoup de transistors. Au final, il n'est pas rare que les concepteurs de processeurs laissent des dépendances structurelles dans leurs pipeline pour éviter de faire grossir leur processeur plus que raison.

Dépendances de données

Cette fameuse ressource à partager peut aussi être un emplacement mémoire. Dans ce cas, on parle de dépendance de données. Deux instructions ont une **dépendance de donnée** quand elles accèdent (en lecture ou écriture) au même registre ou à la même adresse mémoire. Suivant l'ordre dans lequel on effectue des instructions ayant une dépendance de données, le résultat peut changer. Ainsi, un programme devra absolument suivre l'ordre d'accès à la mémoire imposé par le programmeur (ou le compilateur), histoire de donner le bon résultat.

Types de dépendances de données

Différents cas se présentent alors, suivant que les deux instructions écrivent ou lisent cette donnée. La localisation de la donnée n'a pas d'importance : celle-ci peut être en RAM ou dans un registre, on s'en moque ! On se retrouve alors avec quatre possibilités :

Dépendance de données	Effets
Read after Read	Nos deux instructions doivent lire la même donnée, **mais pas en même temps** ! Dans ce cas, on peut mettre les deux instructions dans n'importe quel ordre, cela ne pose aucun problème.
Read after write	La première instruction va écrire son résultat dans un registre ou dans la RAM, et un peu plus tard, la seconde va lire ce résultat et effectuer une opération dessus. La seconde instruction va donc manipuler le résultat de la première.
Write after Read	la première instruction va lire un registre ou le contenu d'une adresse en RAM, et la seconde va écrire son résultat au même endroit un peu plus tard. Dans ce cas, on doit aussi exécuter la première instruction avant la seconde.
Write after	Nos deux instructions effectuent des écritures au même endroit : registre ou adresse mémoire. Dans ce cas aussi, on doit conserver l'ordre des instructions et ne pas réordonnancer, pour les mêmes raisons que les deux

Write	dépendances précédentes.

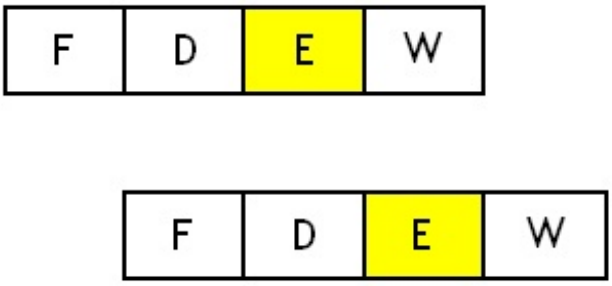
Je dois apporter une petite précision : quand je parle d'ordre des instruction, il y a une petite subtilité. L'ordre en question est celui des lectures et écritures de nos instruction, pas l'ordre de démarrage des instructions. Si deux instructions ont une dépendance de donnée, la première doit avoir terminé sa lecture ou écriture avant que l'autre n'effectue sa lecture ou écriture. Par exemple, si deux instructions ont une dépendance RAW, la première doit avoir écrit son résultat avant que l'autre ne doive le lire. Généralement, cela veut dire que la première instruction doit avoir terminé son exécution avant l'autre. C'est une contrainte assez forte.

Et cette contrainte n'est pas forcément respectée sur un processeur avec un pipeline. Après tout, le principe même du pipeline est de démarrer l'exécution d'une nouvelle instruction sans attendre que la précédente soit terminée. Dans ces conditions, l'ordre de démarrage des instructions est respectée, mais pas l'ordre des lectures et écritures. Cela pose problème avec les dépendances RAW, WAW, et WAR.

Dépendances RAW

Premier problème : les dépendances RAW. Pour utiliser le pipeline de façon optimale, les opérandes d'une instruction ne doivent pas dépendre du résultat d'une instruction précédente. Il serait en effet bien difficile d'exécuter une instruction alors que les données nécessaires à son exécution manquent.

Prenons un exemple simple : on va prendre un pipeline à 4 étages pour se simplifier la vie : Fetch, Decode, Exec, WriteBack. Deux instructions qui se suivent sont chargées l'une après l'autre dans notre pipeline. La seconde instruction doit attendre le résultat de la première pour commencer son exécution. J'ai colorié sur le schéma en jaune les étapes des instructions durant lesquelles l'unité de calcul est utilisée. La situation idéale serait celle-ci.



Mais la réalité est plus cruelle : le résultat de notre instruction n'est disponible qu'après avoir été enregistré dans un registre, soit après l'étape de *Writeback*. Si on ne fait rien, la seonde instruction ne lira pas le résultat de la première, mais l'ancienne valeur qui était présente dans le registre. En clair : le résultat ne sera pas bon !

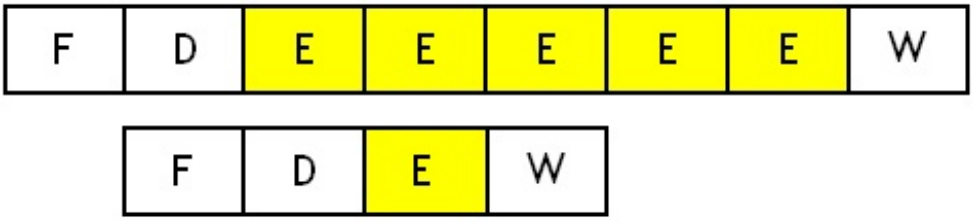
Une solution pour éviter tout problème pourrait être faire en sorte que notre instruction fournisse son résultat en un seul cycle d'horloge. On se retrouverait alors avec un Back End d'un seul cycle pour toutes les instructions. Avec un tel pipeline, les dépendances RAW deviennent impossibles, vu que le résulta est fourni presque immédiatement. Mais c'est impossible à réaliser en pratique, surtout en ce qui concerne les instructions d'accès mémoire ou les instructions complexes.

Dépendances WAW et WAR

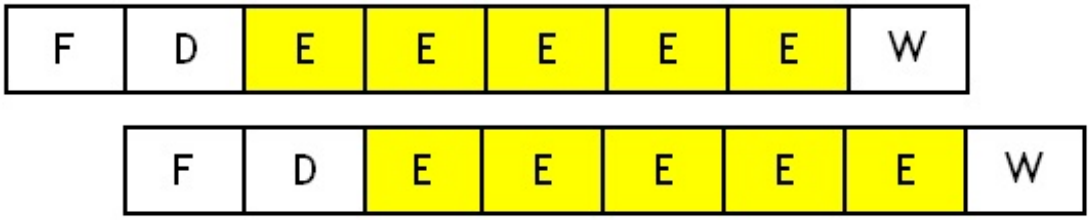
Autre type de dépendances: les dépendances WAR et WAW. A première vue, ces dépendances semblent assez compliquées. Et pour être franc, c'est le cas. Pour commencer, ces dépendances n'apparaissent pas dans tous les pipelines. Certains pipelines bien conçus ne peuvent pas mener à des problèmes avec ces dépendances.

Pour qu'il puisse exister une dépendance WAW, le pipeline doit :

- soit autoriser les instructions multicycles :



;
- soit avoir une écriture qui prend plusieurs étages à elle toute seule :

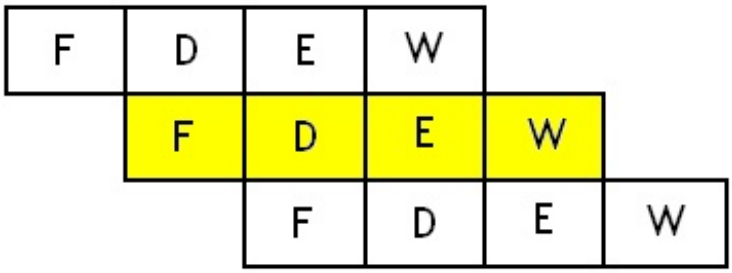


Si aucune de ces conditions n'est réunie, ces dépendances WAW n'apparaissent pas.

Pour les dépendances WAR, c'est un peu la même chose. Elle n'apparaissent que sur certains pipelines, dans lesquels l'écriture des résultats a lieu assez tôt (vers le début du pipeline), et les lectures assez tard (vers la fin du pipeline).

Que faire ?

Pour éviter tout problème avec ces dépendances, on est obligé d'insérer des instructions qui ne font rien entre les deux instructions dépendantes. Dans le schéma qui suit, l'instruction qui ne fait rien est en jaune.



Le seul problème, c'est qu'insérer ces instructions n'est pas trivial. Si on n'en met pas suffisamment, on risque de se retrouver avec des catastrophes. Et si on en met trop, c'est une catastrophe en terme de performances. Déduire le nombre exact d'instruction inutiles à ajouter nécessite de connaitre le fonctionnement du pipeline en détail. Autant dire que niveau portabilité, c'est pas la joie ! La meilleure des solution est encore de déléguer cet ajout d'instructions inutiles au processeur. Et avec quelques circuits en plus, il en est parfaitement capable ! Voyons en détail cette histoire.

Pipeline Bubble / Stall

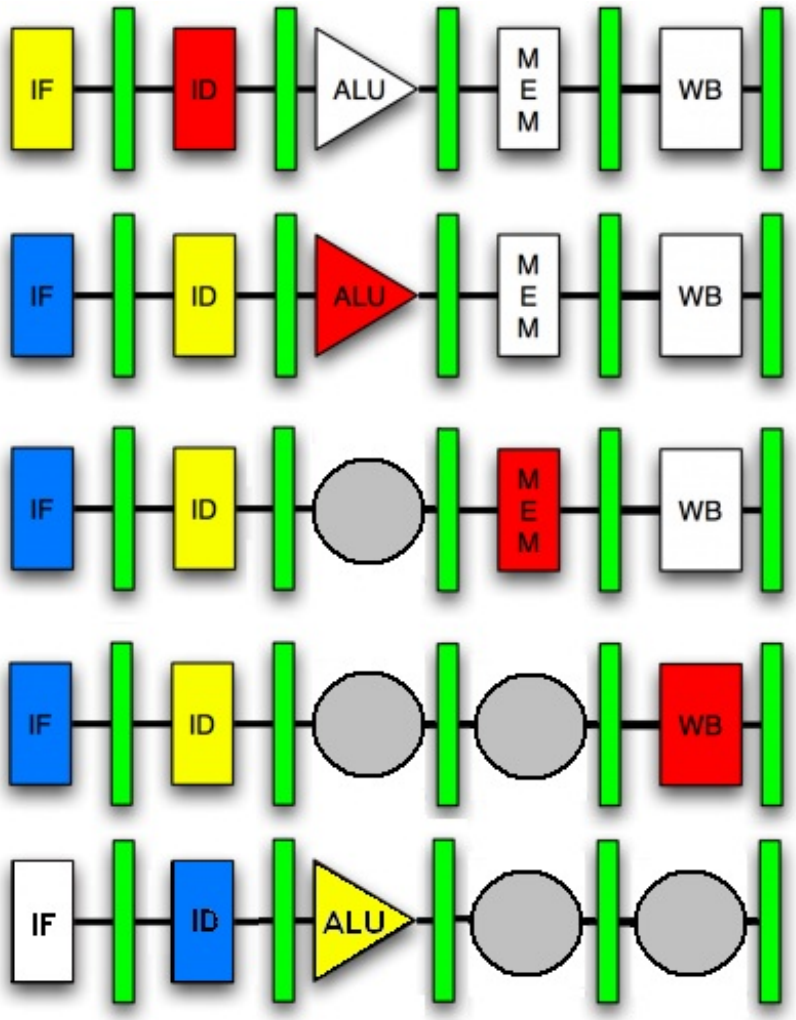
Comme vous le voyez, ces dépendances vont imposer un certain ordre d'exécution pour nos instructions. Cet ordre est un ordre qui est imposé : exécutez les instructions dans un ordre différent, et votre programme donnera un résultat différent. Prenons le cas d'une dépendance RAW : on a une lecture suivie par une écriture. Si on décide de changer l'ordre des deux accès mémoires et que l'on effectue l'écriture avant la lecture, la lecture ne renverra pas la valeur présente avant l'écriture, mais celle qui a été écrite. Toutes les lectures de notre registre ou adresse mémoire précédant notre écriture devront être terminées avant de pouvoir lancer une nouvelle écriture.

A cause de ces dépendances, certaines instructions doivent attendre que toutes les instructions avec lesquelles elles ont une dépendance se terminent avant de pouvoir s'exécuter. Reste à mettre nos instructions en attente.

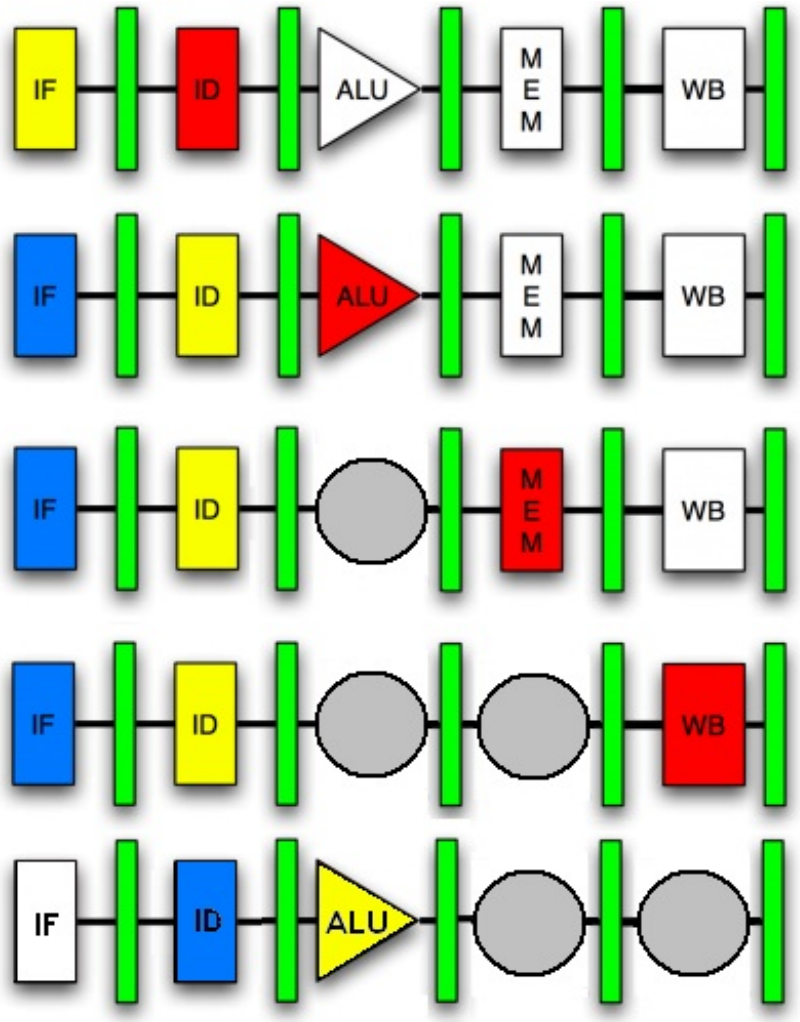
Principe

Le principe est simple : dans le cas le plus simple, tout se passe dans les unités de décodage. Si une dépendance de donnée est détectée, l'unité de décodage d'instruction se bloque tant que la bonne donnée n'est pas disponible. Par bloquer, on veut dire qu'il n'envoie pas l'instruction qu'il vient de décoder aux ALU. Elle met cette instruction en attente, et attend que la donnée à l'origine de la dépendance soit disponible.

Voici ce que cela donne dans notre pipeline. Sur ce schéma, chaque couleur représente une instruction. On remarque qu'en plus de bloquer l'unité de décodage, il faut aussi bloquer tous les étages précédents.



Durant ce temps d'attente, on se retrouvera avec des vides dans le pipeline : certains étages seront inoccupés et n'auront rien à faire. Ce sont les étages marqués en gris sur le schéma suivant.



Ces vides, on les appelle des *Pipeline Stall*, ou encore des ***Pipeline Bubble***. Il apparaissent à chaque fois qu'une instruction doit attendre dans un étage du pipeline.

Et ces *Pipeline Bubble*, ces vides dans le pipeline, sont autant d'étages gaspillés à ne rien faire, qui n'exécutent pas d'instructions. En clair : lorsqu'une *Pipeline Bubble* apparait, notre pipeline est sous-utilisé, et on perd en performances bêtement.

Processeurs In-order

Alors certes, on verra dans ce chapitre et dans la suite du tutoriel, qu'il y a des techniques pour remplir ces vides. Dans la suite du tutoriel, on verra que certains processeurs sont capables de changer l'ordre des instructions pour remplir ces vides. Mais pour le moment, nous allons parler des processeurs qui ne remplissent pas ces vides en changeant l'ordre des instructions. Nous allons commencer par parler des processeurs qui exécutent les instructions dans l'ordre imposé par le programme. Ces processeurs sont appelés des **processeurs *In-Order***.

Voyons un peu comment gérer les différentes dépendances sur ce genre de processeurs.

Dépendances RAW

Les dépendances RAW sont les dépendances les plus évidentes. Vi comment est concu notre pipeline, il est possible qu'une instruction chargée dans le pipeline aille lire une ancienne version d'une donnée. Il suffit pour cela que la version la plus récente n'aie pas encore été écrite par une instruction précédente : cela arrive si celle-ci est encore dans le pipeline. Pour gérer les dépendances sur les processeurs *In-Order*, il suffit simplement d'éviter cela.

Instructions multicycles

Une autre source de dépendances vient de certaines instructions. Il n'est pas rare que certaines instructions monopolisent une unité de calcul durant plusieurs cycles d'horloge. Par exemple, une division peut prendre jusqu'à 80 cycles d'horloge, et rester dans son ALU durant tout ce temps. Et cette ALU est inutilisable pour une autre instruction. On est donc en face d'une belle

dépendance structurelle, qu'il faut résoudre. De même, il est possible que des dépendances WAW apparaissent.

Pour éviter tout problème, l'ALU va devoir fournir des informations l'unité de décodage, qui permettront à celle-ci de décider si elle doit insérer une *Pipeline Bubble* ou non. Notre unité de décodage devra connaitre l'utilisation de chaque unité de calcul (occupée ou non), afin de décider.

Accès à la mémoire

De plus, cette technique pose un problème lors des instructions d'accès à la mémoire. Ces instructions n'ont pas une durée bien définie. La donnée à charger peut se trouver aussi bien dans la mémoire cache, que dans la mémoire RAM, et les temps d'accès ne sont pas les mêmes. On pourrait éventuellement insérer des Pipeline Bubbles dans notre pipeline tant que la lecture ou écriture n'est pas terminée. C'est très simple à implémenter.

- Première implémentation : si l'unité de décodage envoie une instruction qui accède à la mémoire dans le pipeline, elle se bloque, et attend que l'unité qui accède à la mémoire lui envoie un signal électrique pour lui dire : débloque toi, j'ai fini.
- Seconde implémentation : l'unité chargée d'accéder à la mémoire va bloquer elle-même tous les étages précédents du pipeline si elle doit accéder à la mémoire, et elle les libère une fois la donnée lue ou écrite.

Mais dans les deux cas, ce serait un vrai désastre en terme de performances. En tout cas, on verra que ce genre de problème peut facilement être évité avec certaines structures de données matérielles. Mais le moment de parler des *ReOrder Buffers* n'est pas encore arrivé.

Implémentation

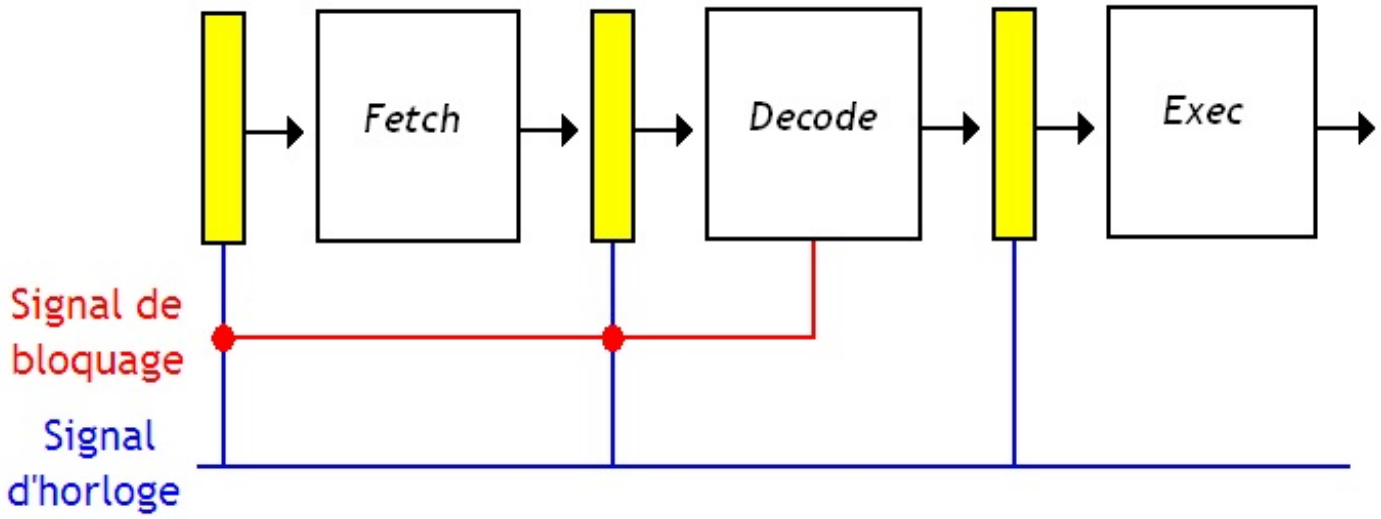
Ces temps d'attente n'apparaissent pas comme pas magie. Il faut concevoir notre processeur pour qu'il fasse ce qu'il faut.

Détection des dépendances

Première chose à faire : détecter les dépendances. Pour cela, l'unité de décodage va comparer chaque registre lu par l'instruction décodée, et les registre de destination des instructions encore dans le pipeline. Si il n'y a aucune correspondance, c'est que l'instruction qu'il vient de décoder n'a pas de dépendances RAW avec les instructions en cours d'exécution. En clair : pas de temps d'attente. Mais dans le cas contraire, l'unité de décodage va bloquer toute les unités précédentes, ainsi qu'elle même, tant que l'instruction n'est pas prête à être exécutée.

Stall

Ce blocage se fait simplement en empêchant l'horloge d'arriver aux registres présents entre les étages de *Fetch et de Décodage*. Ainsi, les registres entre les étages ne seront pas mis à jour, et garderont leur valeur qu'ils avaient au cycle précédent. Cette inhibition de l'horloge se fait en activant un simple interrupteur ou en utilisant un vulgaire petit circuit à base de portes logiques.



Après avoir bloqué les étages précédents, l'unité de décodage va ensuite insérer une instruction `nop` dans le pipeline, histoire que la *Pipeline Bubble* ne fasse rien.

Bypass et Forwarding

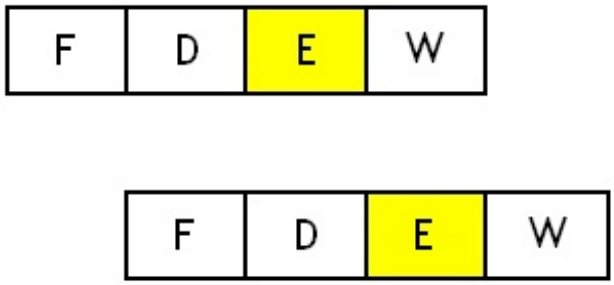
Comme on l'a vu plus haut, les dépendances posent de sacrés problèmes. Alors certes, on peut profiter du fait que nos

instructions soient indépendantes pour les éxecuter en parallèle, dans des étages différents du pipeline, mais cela ne résout pas le problème dus à nos dépendances. Diminuer l'effet des dépendances sur notre pipeline est une bonne idée. Et pour ce faire, diverses techniques ont étés inventées. La première de ces technique s'occupent de diminuer l'impact des dépendances de type RAW. Cette dépendances indique qu'une instruction a besoin du résultat d'une autre instruction pour pouvoir s'exécuter.

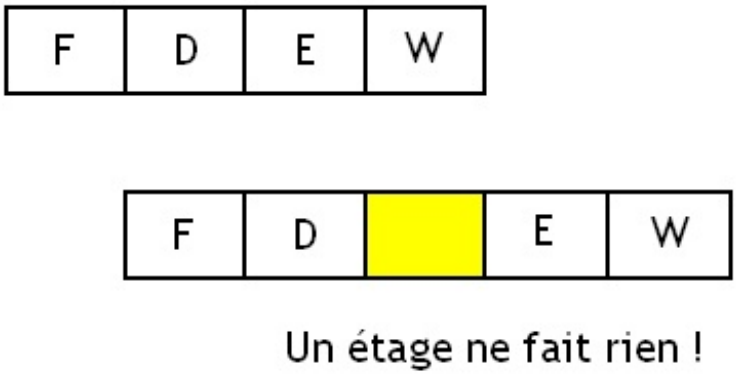
Effet des dépendances RAW

Prenons deux instructions qui se suivent et qui possèdent une dépendance RAW. J'ai colorié sur le schéma en jaune les étapes des instructions durant lesquelles l'unité de calcul est utilisée.

La situation idéale serait celle-ci.



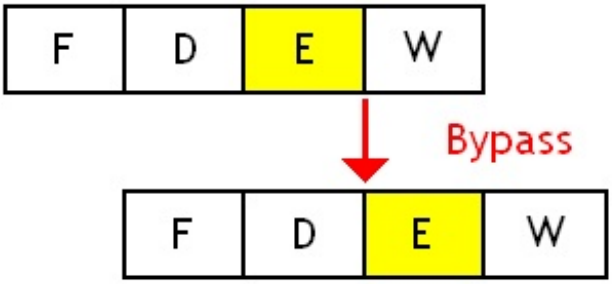
Mais la réalité est plus cruelle : notre pipeline, tel qu'il est actuellement, possède un léger défaut qui peut créer ces temps d'attentes inutiles. Avec le pipeline tel qu'il est actuellement, on doit attendre qu'une instruction soit totalement terminée pour pouvoir réutiliser son résultat. Ainsi, le résultat de notre instruction n'est disponible qu'après avoir été enregistré dans un registre, soit après l'étape de *Writeback*, notée W sur le schéma.



On voit bien que la seconde instruction doit attendre l'autre. Pourtant, son résultat est calculé bien avant que l'instruction ne termine, mais il n'est pas réutilisable immédiatement. Et pour cause : il doit être enregistré dans les registres, et cela ne se fait pas sans précautions. Sans précautions signifie que certaines modifications faites sur les registres ne doivent pas être enregistrées. Par exemple, on ne doit pas enregistrer les résultats calculés après un branchement mal prédit. Il faut aussi gérer les exceptions et les interruptions. Par exemple, si une exception a eu lieu dans le pipeline, on ne doit pas enregistrer les modifications faites à tord par les instructions exécutées après l'exception. Pour cela, il y a des étages qui permettent (ou non) d'enregistrer un résultat en mémoire ou dans les registres.

Bypass

Une idée vint alors : pourquoi ne pas faire en sorte que le résultat d'une l'instruction soit le plus rapidement disponible ? Pas besoin d'attendre que l'instruction termine pour avoir accès au résultat : dès qu'un résultat est calculé, il vaut mieux qu'il soit utilisable par d'autres instructions le plus vite possible.



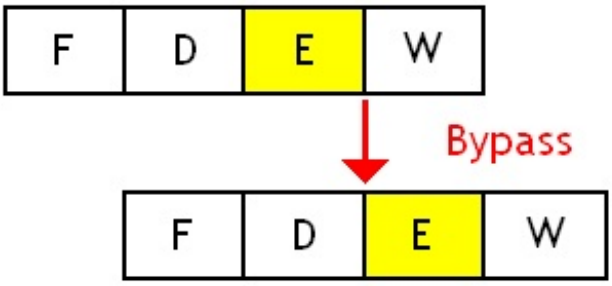
Et si jamais notre instruction réutilise un résultat qui n'est pas bon, il suffit de ne pas enregistrer son résultat.

Quelles instructions ?

Mais cela n'est pas possible pour toutes les instructions. Généralement, il y a deux types d'instructions qui donnent un résultat : les instructions de lecture en mémoire, et les instructions de calcul. Quand on charge une donnée depuis la mémoire, le résultat n'est autre que la donnée lue depuis la mémoire. Et pour les instructions de calcul, il s'agit simplement du résultat du calcul. Pour les accès mémoires, on peut limiter la casse en utilisant des mémoires caches, en jouant sur le pipeline, et utilisant diverses techniques plus ou moins évoluées. Mais On verra cela plus tard. Toujours est-il que ces accès mémoire ont tendance à être très longues, ce qui crée des gros vides dans notre pipeline, difficile à supprimer. Mais on peut toujours jouer sur les instructions de calculs : celle-ci ont des temps d'attente plus faibles, sur lesquels il est facile de jouer.

Principe

D'ordinaire, ce calcul est effectué par notre unité de calcul, et est ensuite écrit dans un registre ou en mémoire après un certain temps. Pour réduire le temps d'attente, on peut faire en sorte que le résultat de notre instruction, disponible ne sortie de l'unité de calcul, soit directement réutilisable, sans devoir attendre qu'il soit écrit dans un registre ou dans la mémoire.



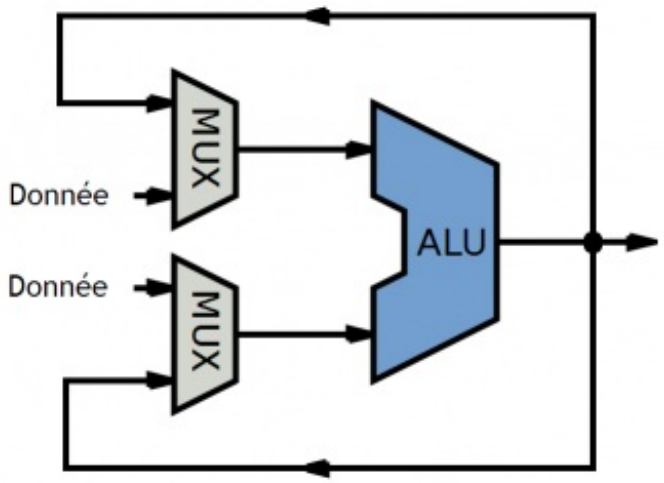
Pour cela, on a juste à modifier les circuits du processeur. Ce résultat peut être directement accédé par la seconde instruction sans devoir être enregistré dans un registre. Ainsi, la seconde instruction peut donc s'exécuter avant que la première aie terminé son étape de *Writeback*. Il s'agit de la technique du ***bypass***, aussi appelée *forwarding*.

Implémentation

Implémenter la technique du bypass dans un processeur n'est pas tellement compliqué sur le principe. Il suffit simplement de relier la sortie des unités de calcul sur leur entrée quand une instruction a besoin du résultat de la précédente. Il faut pour cela ajouter quelques circuits chargé de décider si le contenu de la sortie de l'unité de calcul peut être envoyé sur son entrée.

Multiplexeurs

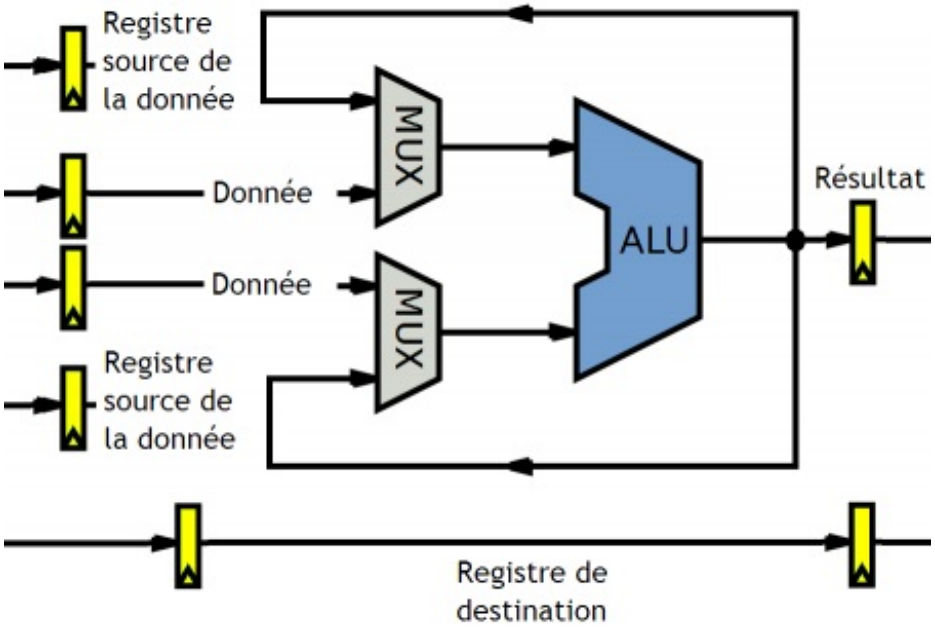
Cela se fait en utilisant des multiplexeurs : on relie la sortie de notre ALU à un multiplexeur dont la sortie sera reliée à l'entrée de cette même unité de calcul, l'autre entrée étant reliée au bus interne au processeur. On pourra ainsi relier (ou pas) la sortie de l'ALU à une de ses entrées en configurant ce multiplexeur.



Identification des registres

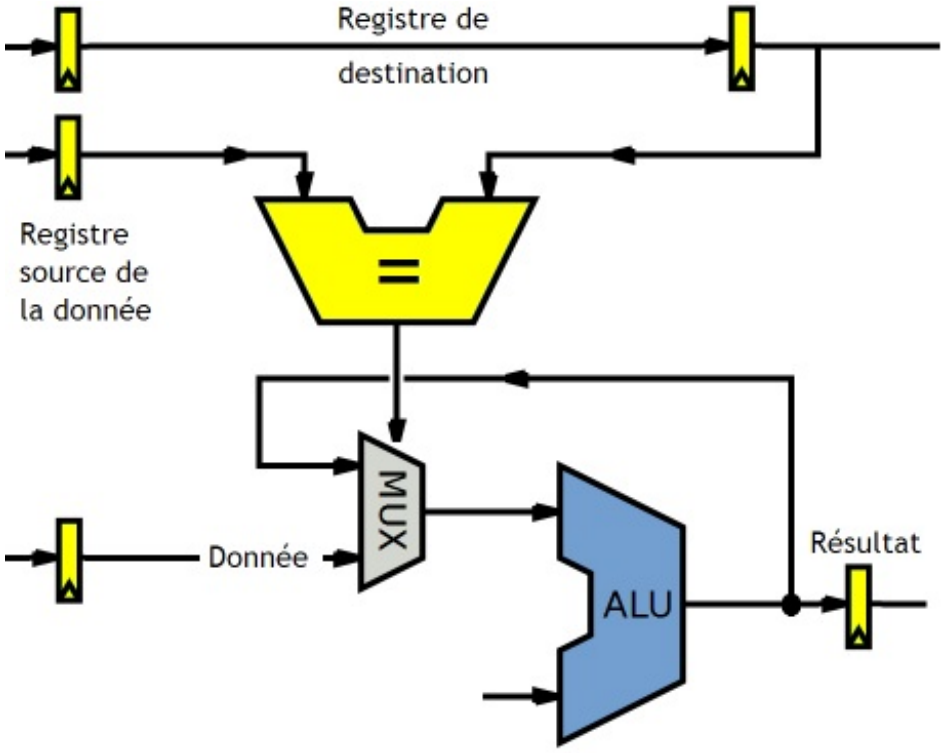
En faisant comme ceci, il suffit alors de commander le multiplexeur en utilisant son entrée de commande pour que celui-ci recopie ou non la sortie de l'ALU sur son entrée. Reste à savoir quoi mettre sur cette entrée de commande. Pour cela, on doit détecter les situations dans lesquelles un résultat placé sur la sortie de l'ALU doit être envoyée sur l'entrée. En clair, détecter une dépendance RAW. Cela arrive quand le registre modifié par une instruction est le même que le registre lu par la suivante.

Pour connaitre ces registres, rien de plus simple. Il suffit de propager leurs noms dans le pipeline, avec les autres signaux de commandes. Ainsi, ces noms de registres vont passer d'un étage à un autre, avec les autres aignaux de commandes de leur instruction, et seront donc utilisables au besoin.



Forwarding Unit

Pour détecter nos dépendances, il suffira de comparer le registre destination dans le registre placé après l'ALU, et le registre source placé en entrée. Si ce registre est identique, on devra faire commuter le multiplexeur pour relier la sortie de l'ALU sur l'entrée. Cette comparaison est effectuée par deux comparateurs, chacun commandant un multiplexeur.



Clusters

Sur les processeurs ayant beaucoup d'unités de calculs, on trouve souvent un système de *bypass* (*forwarding*) assez étoffé. Après tout, rien ne dit que deux instructions dépendantes seront réparties sur la même unité de calcul. Le *Scheduler* peut très bien envoyer deux instructions ayant une dépendances RAW sur des unités de calcul différentes. Pour cela, il n'y a qu'une solution : chaque sortie d'une unité de calcul doit être reliée aux entrées de toutes les autres ! Et bien sûr, on doit aussi rajouter les multiplexeurs et comparateurs qui vont avec. Je ne vous raconte pas la difficulté à câbler un réseaux d'interconnexions pareil. Pour n unités de calcul, on se retrouve avec pas moins de n^2 interconnexions. Autant dire que cela fait beaucoup de fils et que sur certains processeurs, cela peut poser problème.

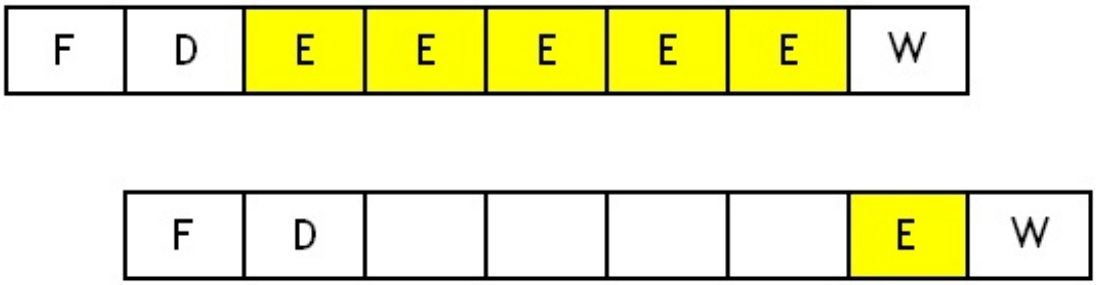
Sur certains processeurs, on arrive à se débrouiller sans trop de problèmes. Mais quand on commence à mettre un peu trop d'unités de calculs dans le processeur, c'est terminé : on ne peut pas forcément, ou l'on a pas envie, de câbler un réseau de *bypass* aussi touffu et complexe. Pour limiter la casse, on utilise alors une autre méthode : on ne relie pas toutes les unités de calcul ensemble. A la place, on préfère regrouper ces unités de calcul dans différents blocs séparés qu'on appelle des *clusters*. Le bypass est alors rendu possible entre les unités d'un même *cluster*, mais pas entre les unités appartenant à deux *clusters* différents.
Toutes ces dépendances vont imposer un certain ordre d'exécution à nos instructions. Par chance, cet ordre ne permet pas d'ordonner toutes les instructions : autant dans l'ordre d'exécution imposé par le programme toutes les instructions sont ordonnées et doivent s'exécuter les unes après les autres, autant l'ordre des dépendances laisse plus de libertés. Si deux instructions n'ont pas de dépendances de données, c'est qu'elles sont indépendantes. Dans ce cas, le processeur peut les exécuter en parallèle ou changer leur ordre d'exécution sans problème. Il existe diverses techniques pour ce faire : Exécution *Out Of Order*, Superscalarité, etc. Mais pour le moment, voyons un peu quel est l'effet de ces dépendances sur notre pipeline, et comment on peut limiter la casse.

Execution Out Of Order

On l'a vu, nos instructions multicyles entrainent d'apparition de dépendances structurelles et de dépendances de données assez embêtantes. Si deux instructions indépendantes sont chargées à la suite dans le pipeline, pas de problèmes. Mais dans le cas contraire, on va se retrouver avec un problème sur les processeurs *In-Order*. Les dépendances vont créer des temps d'attentes. De nombreuses techniques ont étés inventées pour supprimer ces temps d'attentes, dont le *Bypassing*. Mais cette dernière ne suffit pas toujours. Pour supprimer encore plus de temps d'attentes, on a inventé ***l'exécution Out Of Order*** . Voyons en quoi celle-ci consiste.

Principe

Introduisons le problème par un exemple. Prenons deux instructions qui se suivent et qui possèdent une dépendance de données. La seconde devant attendre le résultat de la première pour commencer son exécution. J'ai colorié sur le schéma en jaune les étapes des instructions durant lesquelles l'unité de calcul est utilisée.



On voit bien que la seconde instruction doit attendre l'autre. C'est un fait : si deux instructions ont une dépendance de donnée et que celle qui précède l'autre est une instruction multicycle, alors on va perdre du temps dans ce fameux temps d'attente. Des problèmes similaires apparaissent lors des accès mémoires. Si une donnée attend des données en provenance de la mémoire, on se retrouve aussi avec ce genre de vides, sauf que l'unité de calcul est libre.

Les deux types de dépendances, structurelles ou de données, ainsi que les accès mémoires, on exactement le même effet sur notre pipeline : ils forcent certains étages du pipeline à attendre qu'une donnée ou qu'un circuit soit disponible. Dans ce cas, on est obligé de modifier légèrement le comportement de notre pipeline : on doit stopper l'étage qui attend et l'immobiliser, ainsi que celui de tous les étages précédents. En clair, notre étage ne fait rien et attend sagement, sans passer à l'instruction suivante. Et les étages précédents font pareils. C'est la seule solution pour éviter les catastrophes. Et pour cela, on doit rajouter des circuits qui vont détecter les dépendances dans notre pipeline et qui vont stopper le fonctionnement de l'étage devant attendre ainsi que les étages précédents.

C'est le résultat des dépendances de donnée : certains étages ne font rien, pendant que certaines instructions attendent la donnée voulue. L'instruction va devoir attendre que ses opérandes soient disponibles dans un étage et va **bloquer tous les étages précédents dans le pipeline** : cela retarde le chargement des instructions suivantes. Au final, notre pipeline exécute moins d'instructions en simultané que prévu.

Une idée géniale

Un jour, quelqu'un a eu une idée pour éviter de perdre du temps dans ces vides, et a décidé que ce temps d'attente pouvait être remplis par des instructions indépendantes. Au lieu d'exécuter des instructions dans l'ordre du programme, il suffit simplement de changer l'ordre des instructions du programme pour remplir ces vides avec des instructions indépendantes. On profite ainsi du fait que l'ordre induit par les dépendances de données est moins strict que l'ordre imposé par le programmeur et par le *Program Counter*. Et pourtant, ces deux ordres donnent lieu au même programme, qui effectue les mêmes choses.

Les moyens pour répartir correctement nos instructions pour combler les vides sont assez restreints, et il n'en existe plusieurs. On peut par exemple stocker nos instructions dans la mémoire en incorporant à l'intérieur des informations sur leurs dépendances de donnée. Cela permet de laisser le processeur gérer ces dépendances tout seul et décider de l'ordre dans lequel éxecuter nos instructions tout seul. Il peut ainsi paralléliser automatiquement un programme et éxecuter ses instructions dans l'ordre des dépendances de données tout seul. C'est ce qui se fait sur les architectures *dataflow*, mais cela n'est pas utilisé sur les processeurs modernes. Les architectures *dataflow* ont en effet divers défauts qui plombent leurs performances.

OOO et processeurs séquentiels

A la place, on part d'un programme purement séquentiel, dans lequel les instructions s'exécutent les unes après les autres. Et on change l'ordre des instructions pour remplir les vides. Ce ré-ordonnancement peut se faire de deux façons différentes : soit on le fait à la compilation, soit c'est le processeur qui va déterminer les dépendances entre instructions et qui va changer lui-même l'ordre des instructions du programme. Quand c'est le processeur qui s'en occupe, aucune information ne lui est fournie

explicitement dans les instructions (comme c'est le cas sur les architectures *dataflow*), et il doit déduire les dépendances entre instructions tout seul, en regardent les registres ou adresses mémoires utilisées par l'instruction. Une fois ces dépendances connues, il change lui-même l'ordre des instructions et décide de leur répartition sur les différentes unités de calcul.

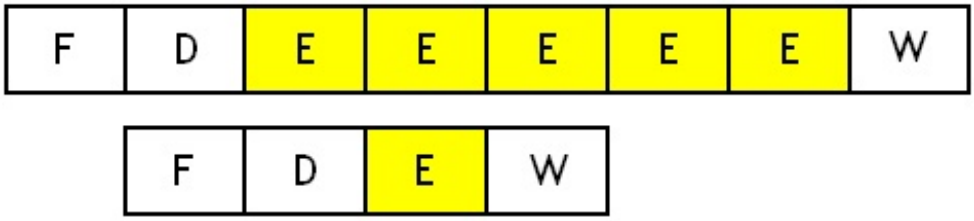
Chacune de ces solutions a ses avantages et ses inconvénients. Par exemple, le compilateur peut travailler sur des morceaux de programmes plus gros et ainsi réordonnancer les instructions un peu mieux que le processeur, qui ne traitera qu'une dizaine ou une vingtaine d'instruction à la fois. Mais par contre, le processeur peut virer certaines dépendances qui apparaissent à l'exécution (branchements) : cela est impossible pour le compilateur. Bien sûr, rien n'empêche de faire les deux, pour avoir un maximum de performances.

La dernière solution qui consiste à réorganiser l'ordre dans lequel s'exécutent les instructions du programme pour gagner en efficacité s'appelle **l'exécution *Out-Of-Order***. Avec cette solution, si une instruction doit attendre que ses opérandes soient disponibles, on peut remplir le vide crée dans le pipeline par une autre instruction indépendante de la première : on exécute une instruction indépendante au lieu de ne rien faire. Cette instruction pourra s'exécuter avant l'instruction qui bloque tout le pipeline et n'aura donc pas à attendre son tour : on exécute bien des instructions dans le désordre.

Petit détail : quand je parle d'instructions, il peut s'agir aussi bien d'instructions machines (celle que notre processeur fourni au programmeur), que de micro-instructions (celles qu'on a vue dans le chapitre sur la micro-architecture d'un processeur, quand on a parlé de séquenceurs micro-codés). Si votre processeur utilise un séquenceur micro-codé, la réorganisation des instructions se fera non pas instructions machines par instructions machines, mais micro-opérations par micro-opérations. Cela permet d'avoir plus de possibilités de réorganisations.

Une dépendance structurelle à résoudre

Mais pour cela, il lui fallait résoudre un premier problème. L'idée consistant à remplir les vides du pipeline par des instructions indépendantes revient à faire ce qui suit.



Problème : l'unité de calcul n'accepte qu'une seule instruction, et la situation du dessus est impossible. On est donc obligé de retarder l'exécution de notre seconde instruction. On va donc appliquer la solution de base en cas de dépendances structurelle : on dupliquer le circuit fautif, qui est ici l'unité de calcul, et on répartit nos instructions sur les différentes unités de calcul. Le seul problème, c'est que rien n'empêche à une instruction chargée dans le pipeline après une autre de finir avant celle-ci. Les instructions ayant des temps d'exécutions variables, les résultats peuvent arriver dans le désordre. Et dans un cas pareil, il faut absolument gérer correctement les dépendances de données entre instructions correctement.

Deux types d'Out Of Order

Il existe différents types d'*Out-Of-Order*, dont les deux plus simples sont le *scoreboarding*, et l'algorithme de Tomasulo. Ce dernier a été inventée par Robert Tomasulo, un chercheur travaillant chez IBM, et fut utilisé pour la première fois dans la FPU des processeurs IBM 360. Autant dire que cela remonte à loin : 1970 ! Ces deux techniques d'*Out Of Order* sont basées sur un pipeline normal, mais auquel on a ajouté diverses modifications.

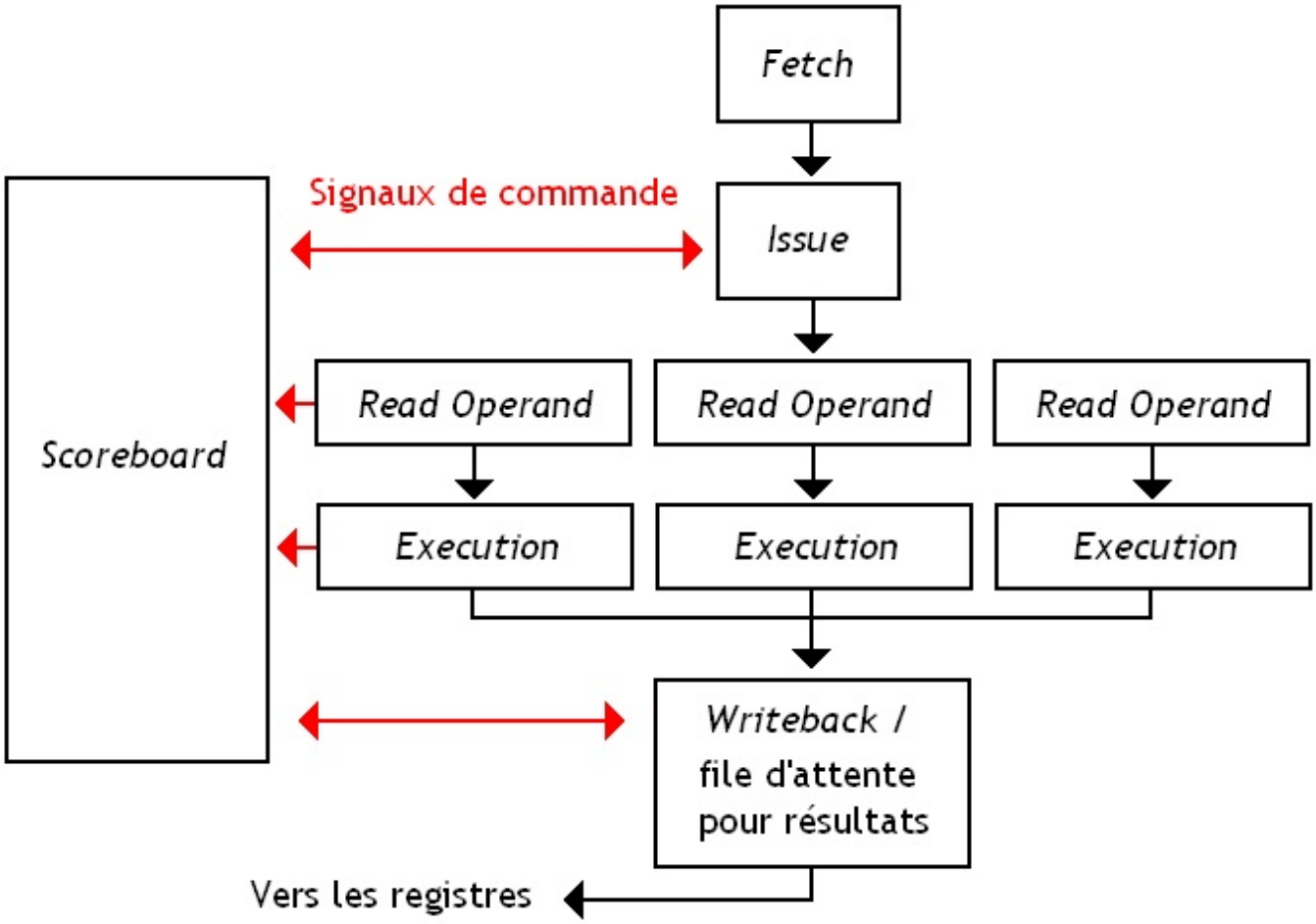
Ces deux techniques ont malgré tout une caractéristique commune : avec elles, les instructions sont décodées les unes après les autres, dans l'ordre du programme, et sont ensuite exécutées dans un ordre différent. Et il faut avouer qu'il est difficile de faire autrement. Qui plus est, on ne décode qu'une instruction à la fois. Dans le prochain chapitre, on verra que certains processeurs arrivent à passer outre cette limitation. Sur ces processeurs, on pourra ainsi décoder plusieurs instructions simultanément sans aucun problèmes. Mais nous n'en sommes pas encore là !

Scoreboarding

Commençons par la première de ces techniques : le ***scoreboarding***. Avec le *scoreboarding*, la gestion des dépendances de données est effectuée par un seul circuit, nommé le *scoreboard* : il se chargera ainsi de gérer l'ensemble des unités de calcul et se chargera de tout ce qui a rapport à l'exécution *Out Of Order*. Son rôle est de détecter les dépendances de données entre instructions et d'exécuter au mieux les instructions en conséquence. Mais ce *scoreboard* n'est pas le seul circuit nécessaire pour pouvoir éxecuter nos instructions dans un ordre différent de celui imposé par le programme.

Pipeline d'un processeur Scoreboardé

Évidemment, rajouter un circuit de ce genre nécessite de modifier quelque peu le pipeline. Il faut bien rajouter des circuits chargés de modifier l'ordre d'exécution de nos instructions. Le pipeline d'un processeur utilisant le *scoreboarding* est illustré par les schéma ci-dessous.



N'ayez pas peur : les explications arrivent !

Étape d'issue

Au moins une étape supplémentaire apparait : l'étape d'***issue***. Cette étape remplace l'étape de décodage de l'instruction, ou plutôt elle la complète. L'étape d'*issue* va non-seulement décoder notre instruction, mais elle va aussi décider si notre instruction peut démarrer son exécution ou non. Pour cela, notre étape d'*issue* va vérifier quels sont les registres utilisés pour stocker le résultat de notre instruction et va vérifier que notre instruction n'a aucune dépendance WAW (elle n'a pas besoin d'écrire son résultat au même endroit qu'une autre instruction qui la précède dans l'ordre du programme). Elle va aussi vérifier si une unité de calcul est disponible pour l'exécuter. Si notre instruction possède une dépendance WAW ou qu'aucune unité de calcul disponible ne peut l'exécuter, alors notre instruction va attendre son tour bien sagement. Dans le cas contraire, elle va pouvoir progresser dans le pipeline : le *scoreboard* sera alors mis à jour.

Attention, toutefois : nos instructions sont décodées et envoyées aux unités de calculs les unes après les autres, en série. On peut éventuellement en envoyer plusieurs (il suffit d'adapter un peu l'étape d'*issue*), mais cet envoi vers les ALU se fera dans l'ordre imposé par le programme. Vu comme cela, on se demande bien où est l'*Out Of Order*. Mais celui-ci vient après, plus loin dans le pipeline.

L'étape Read Operand

Notre instruction n'a donc aucune dépendance WAW, et une unité de calcul est prête à l'exécuter. Visiblement, tout va bien dans le meilleur des mondes, et notre instruction est donc surement bien partie pour fournir un résultat dans pas longtemps. Mais la réalité est un peu plus cruelle : il n'y a certes pas de dépendances WAW, mais les autres dépendances doivent être prises en compte. Les dépendances RAW, notamment : rien n'indique que les données que notre instruction doit manipuler sont prêtes ! Après tout, notre instruction peut très bien attendre une opérande en provenance de la mémoire ou calculée par une autre

instruction. Dans ce cas, notre instruction doit attendre que la donnée en question doit lue depuis la mémoire ou calculée par l'instruction avec laquelle elle partage une dépendance. Il faut donc trouver un moyen pour éviter de commencer à exécuter notre instruction alors que seulement la moitié de ses données sont déjà disponibles.

Mais qu'à cela ne tienne, envoyons-la quand même à l'unité de calcul !

Heu... exécuter une instruction sans avoir ses opérandes, c'est pas un peu casse-cou ?

Si, effectivement. Mais rassurez-vous, les concepteurs de processeurs *Out Of Order* ont pensés à tout (en même temps, on les paye pour ça !).

Pour éviter les problèmes, une seule solution : notre instruction va être mise en attente en attendant que ses opérandes soient disponibles. Pour ce faire, on ajoute une étape dans notre pipeline, gérée par notre *scoreboard*. Cette étape se chargera de retarder l'exécution de notre instruction tant que ses données ne seront pas disponibles. Notre *scoreboard* va ainsi stocker notre instruction dans une petite mémoire tampon (avec éventuellement ses opérandes), et la fera attendre tant que les opérandes ne sont pas prêtes.

Exécution

Une fois ses opérandes prêtes, l'instruction est envoyée à une unité de calcul et est exécutée. C'est à cet endroit du pipeline que l'on peut vraiment parler d'*Out Of Order* : notre processeur contiendra obligatoirement plusieurs unités de calcul, dans lesquelles on pourra éxecuter des instructions en parallèle. Sur un processeur sans *scoreboarding* et sans *Out Of Order*, lorsqu'une instruction n'a pas fini de s'exécuter (si elle prend plusieurs cycles ou attends des données), on doit attendre que celle-ci se termine avant de pouvoir en démarrer une autre. Avec le *scoreboarding*, ce n'est pas le cas : si une instruction prend plusieurs cycles pour s'exécuter ou attend des données, on peut démarrer une nouvelle instruction au cycle suivant dans une autre unité de calcul disponible. Pas besoin d'attendre : une instruction indépendante peut ainsi commencer son exécution au cycle suivant sans attendre la fin de l'instruction précédente.

Évidemment, cela marche uniquement si une instruction prend plus d'un cycle pour s'exécuter : si elles attend des données depuis la RAM, ou si il s'agit d'une instruction complexe demandant beaucoup de temps. Si toutes les instructions mettent un cycle d'horloge pour s'exécuter (accès mémoires compris), le *scoreboarding* ne sert à rien vu qu'on décode nos instructions unes par unes.

L'exécution de l'instruction va prendre un ou plusieurs cycles d'horloge, et on doit absolument prendre en compte cet état de fait : toutes les instructions ne prennent pas le même temps pour s'exécuter. Ainsi, leur résultat sera disponible plus ou moins tardivement, et les *scoreboard* doit en tenir compte. Pour cela, quand le résultat de notre instruction est calculé, le *scoreboard* est prévenu par l'unité de calcul et il se mettra à jour automatiquement.

Writeback

Enfin, le résultat de notre instruction est prêt. Il ne reste plus qu'à le sauvegarder dans un registre ou dans la mémoire...enfin presque ! Ben oui, il reste les dépendances WAR : il faut attendre que toutes les lectures dans le registre que l'on va écrire soient terminées. Pour cela, il n'y a pas vraiment de solution : on doit encore une fois attendre que ces lectures soient finies pour autoriser l'écriture du résultat. Seule solution : placer nos résultats dans une file d'attente, et autoriser l'écriture du résultat dans le *register file* une fois que toutes les lectures de la donnée présente dans le registre à écrire soient terminées. Bien évidemment, c'est le *scoreboard* qui autorisera ou non cette écriture.

Scoreboard

On a vu précédemment que le *scoreboard* a tout de même pas mal de responsabilités : il doit faire patienter les instructions possédant une dépendance WAW dans l'étage d'*issue*, faire patienter notre instruction dans l'étage de *Read Operand* tant que ses données ne sont pas disponibles, et temporiser l'écriture du résultat d'une instruction tant que les lectures précédentes dans le registre à écrire ne sont pas effectuées. Cela nécessite de commander pas mal d'étage du pipeline, ainsi que quelques circuits annexes.

Mine de rien, notre *scoreboard* va devoir avoir accès à pas mal d'informations pour gérer le tout efficacement.
Celui-ci devra retenir :

- les dépendances entre instructions ;
- quelles sont les instructions en cours d'exécution ;
- des unités de calcul chargée de l'exécution de celles-ci ;
- et les registres dans lesquels nos instructions en cours d'exécution vont écrire leur résultat.

Pour cela, notre *scoreboard* va contenir :

- **une table des registres**, qui indiquera dans quel registre stocker le résultat fourni par une unité de calcul (par une instruction, quoi) ;
- **une table stockant l'état de chaque instruction** : est-elle en cours d'exécution, attend-elle ses opérandes, son résultat est-il en attente d'écriture, etc ;
- et une **table d'instruction** qui contient des instructions sur chaque instruction à éxecuter : son opcode, où placer le résultat, dans quels registres sont stockées les opérandes, et est-ce que ces opérandes sont disponibles.

Final

Au final, on s'aperçoit que cette forme d'*Out Of Order* est assez rudimentaire : les instructions sont toujours décodées dans l'ordre du programme et sont réparties sur une ou plusieurs unités de calcul. Par contre, celles-ci sont ensuite réparties sur des unités de calculs séparées dans lesquelles elles peuvent s'exécuter dans un ordre différent de celui imposé par le programme. Pire : elles peuvent enregistrer leurs résultats dans un ordre différent de celui imposé apr le programmeur : elles fournissent leur résultat dès que possible. Néanmoins, on observe quelques limitations assez problématiques : les dépendances RAW, WAW et WAR vont faire patienter nos instructions. Et il serait bien de limiter un peu la casse à propos de ces dépendances !

Out Of Order Issue

Notre *Scoreboard* est un algorithme assez sympathique, capable de fournir de bonnes performances. On peut espérer obtenir des gains proches de 10 à 30 % suivant les programmes. Mais celui-ci a toutefois un léger défaut. Ce défaut n'est pas visible immédiatement. Pour le voir, il faut bien regarder l'étape d'*Issue*. Le problème vient du fait que notre unité d'Issue va envoyer nos instructions à l'étape suivante dans l'ordre du programme. Ces instructions pourront alors s'exécuter dans le désordre, mais leur envoi aux unités de calcul s'effectue dans l'ordre du programme.

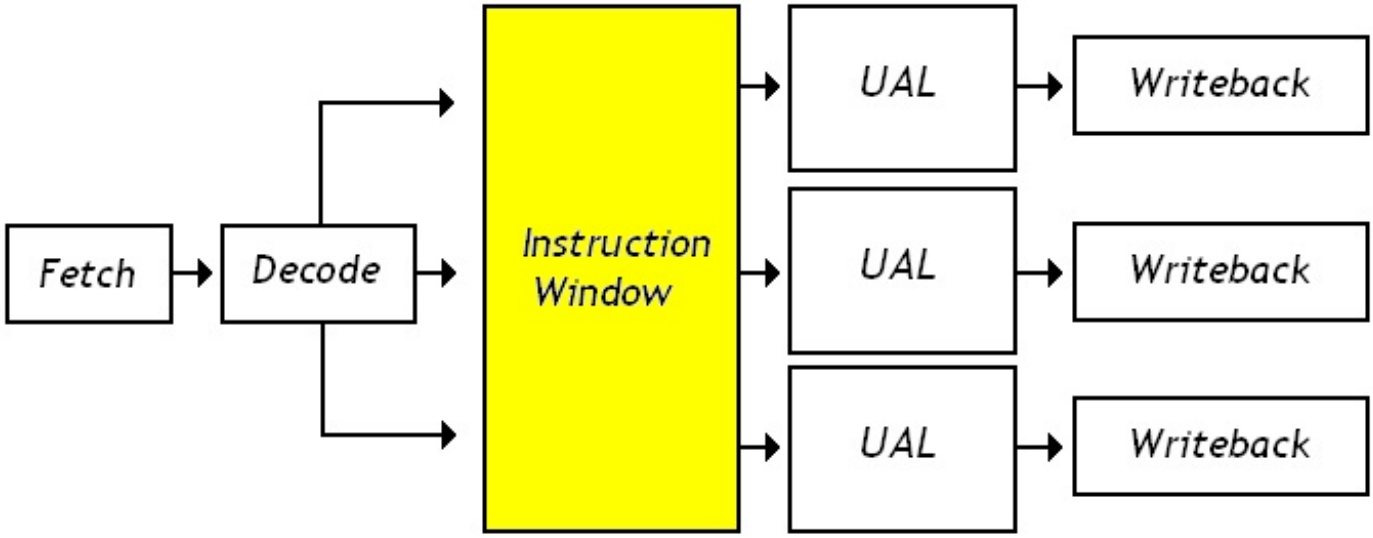
Cela pose un problème si jamais deux instructions se suivent dans notre programme. Si jamais ces deux instructions ont une dépendance structurelle ou une dépendance WAW, l'étape d'*Issue* devra alors insérer des *Pipeline Bubbles* et bloquer les unités de *Fetch* et de *Decode*. C'est nécessaire pour que notre programme fasse ce pour quoi il a été programmé : sans cela, des dépendances WAW seront violées et le programme qui s'exécute ne fera pas ce pour quoi il a été programmé. Limiter ces *Pipeline Bubbles* dues à des dépendances entre instructions est un moyen assez intéressant pour gagner en performances. Encore une fois, nous allons devoir trouver un moyen pour limiter la casse, en améliorant notre algorithme de *Scoreboarding*.

L'idée est très simple : au lieu de bloquer notre pipeline à l'étape d'*Issue* en cas de dépendances, pourquoi ne pas aller chercher des instructions indépendantes dans la suite du programme ? Après tout, si jamais j'ai une dépendance entre une instruction I et une instruction I+1, autant aller regarder l'instruction I+2, voire I+3. Si cette instruction I+2 est indépendante de l'instruction I, je pourrais alors *Issue* l'instruction I+2 dans mon pipeline deux cycles plus tard. Cela limite fortement la casse.

Centralized Instruction Window

Pour que cela fonctionne, on est obligé de faire en sorte que notre étape d'Issue puisse mémoriser plusieurs instructions. Dans notre exemple du dessus, si l'instruction I est chargée dans notre pipeline, je devrais mettre en attente l'instruction I+1 dans l'étape d'*Issue*, et y charger l'instruction I+2. Notre étape d'Issue devra pouvoir stocker plusieurs instructions. Cela permettra mettre en attente les instructions ayant une dépendance WAW ou structurelle avec les instructions en cours d'exécution. Nos instructions seront donc chargées dans notre pipeline et décodées les unes après les autres. Elles seront alors mises en attente dans l'étape d'*Issue*, et quitteront celle-ci une fois que toute dépendance problématique sera supprimée.

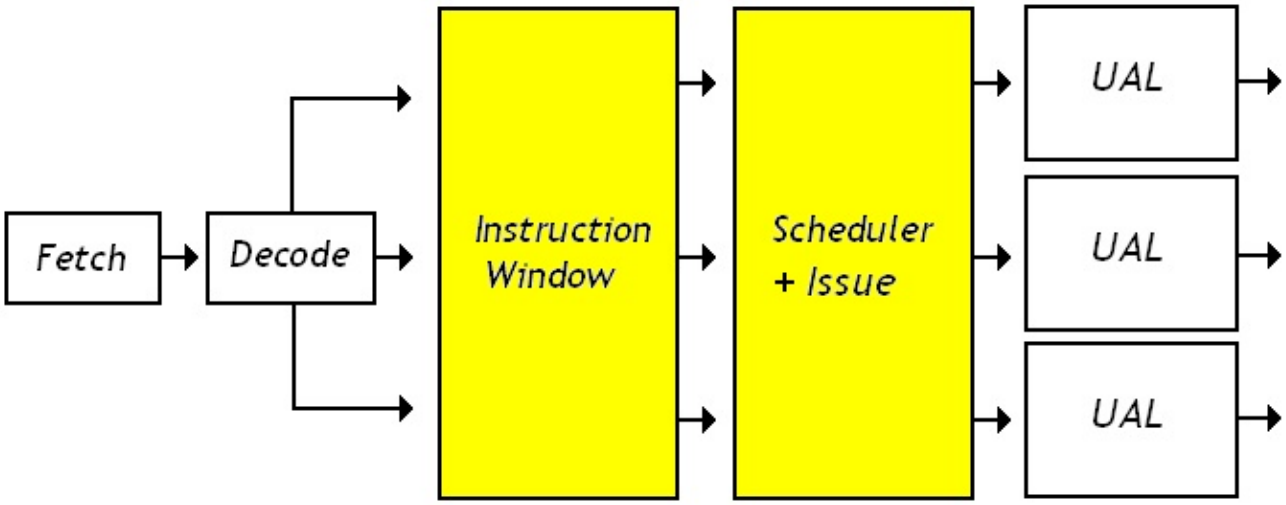
Cette mise en attente se fait grâce à ce qu'on appelle une ***Instruction Window***. Pour cela, les instructions préchargées (et éventuellement décodées) seront placées dans une sorte de mémoire tampon, qui stockera les instructions en attente de traitement. L'unité chargée de répartir les instructions et de vérifier leurs dépendances sera reliée à cette mémoire tampon et pourra donc manipuler ces instructions comme il le souhaite. Cette mémoire tampon s'appelle l'***instruction window***.



Cette *instruction window* est gérée différemment suivant le processeur. On peut par exemple la remplir au fur et à mesure que les instructions sont exécutées et quittent donc celle-ci. Ou on peut attendre que celle-ci soit vide pour la remplir intégralement en une fois. Les deux façons de faire sont possibles.

Vous remarquerez que j'ai placé les unités du séquenceur sur le schéma. Mais le décodeur d'instruction peut être placé de l'autre coté de l'*instruction window*. En fait, celles-ci peuvent se retrouver des deux cotés de l'*instruction window*. On peut en effet décoder les instructions avant leur passage dans l'*instruction window* : c'est ce qui se fait surement dans vos processeurs x86 actuels et dans la majorité des processeurs utilisant la micro-programmation horizontal (pour ceux qui ont oubliés, allez voir ici : le chapitre sur la micro-architecture d'un processeur de ce tutoriel). Mais on peut aussi se débrouiller pour faire autrement, en plaçant les unités de décodage après l'*instruction window*.

Pour choisir quelle instruction éxecuter, il suffira de rajouter un circuit qui se chargera de lire le contenu de cet *Instruction Window*, et qui décidera quelles sont les instructions à exécuter. Ce circuit, le ***Scheduler***, regardera quelles sont les instructions dans cet *Instruction Window* qui sont indépendantes et décidera alors de les répartir sur les unités de calcul inutilisées au besoin.

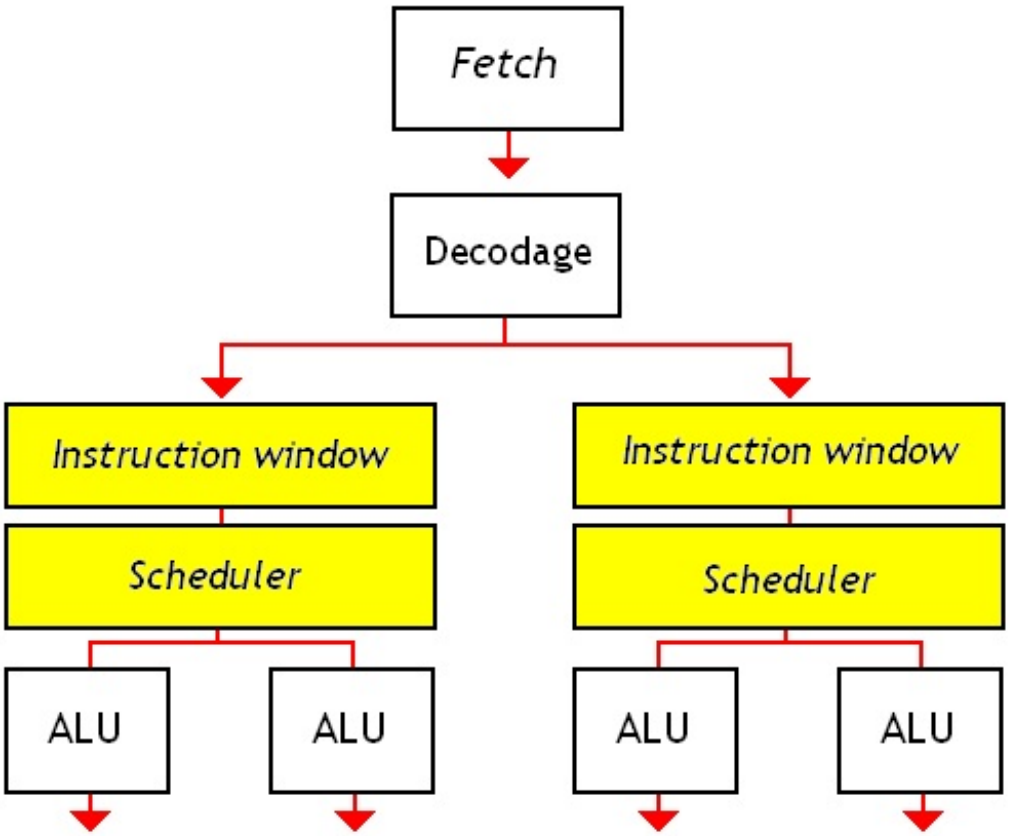


Au fait : ce *Scheduler* n'est rien d'autre que le circuit en charge de l'étape d'*Issue*, auquel on permet de traiter plusieurs instructions. C'est lui qui vérifiera les dépendances entre instructions et qui sera chargé de répartir des instructions sur plusieurs unités de calcul.

Instructions Windows Multiples

Sur certains processeurs, cette gestion des dépendances et la répartition des instructions à l'exécution sur les ALU sont séparées dans deux circuits distincts, séparés par une mémoire tampon. On appelle ***Schelving*** cette séparation entre *Issue* et vérification des dépendances.

Sur un processeur implémentant le *Schelving*, on ne trouve non pas une, mais plusieurs *Instructions Windows*. Généralement, on trouve une *Instruction Window* par ALU. L'unité d'*Issue* va simplement vérifier qu'une unité de calcul est libre pour exécuter notre instruction, et va l'envoyer dans une de ces *Instruction Window*, attachée à la bonne ALU.



Une fois placée dans cette *Instruction Window*, notre instruction va attendre que toutes les dépendances soient résolues avant de s'exécuter. A chaque cycle d'horloge, un circuit va vérifier que notre instruction a ses opérandes de prêtes, que l'unité de calcul est libre, et vérifie aussi les dépendances WAR et WAW. Une fois que notre instruction n'a plus aucune dépendance, elle est directement envoyée à l'ALU et elle s'exécute. Dans le cas contraire, elle patiente dans l'*Instruction Window*. Généralement, ces *Instructions Windows* spécialisées sont parfois appelées des *Reservations Stations*.

Utiliser du *Shelving* a un avantage : les *Schedulers* attachés aux *Reservations Stations* sont souvent beaucoup plus simples et consomment nettement moins de circuits. Plusieurs raisons à cela. Pour commencer, chaque *Reservation Station* est pluis petite qu'une grosse *Instruction Window* : la sélection de la prochaine instruction à envoyer à l'ALU est donc plus simple. Après tout, entre sélectionner une instruction parmi 100, et une parmi 10, un des deux cas est le plus simple, et vous avez deviné lequel. Ensuite, une *Reservation Station* ne peut libérer qu'une entrée par cycle, vu qu'elle envoie une instruction à la fois sur l'ALU. Mais avec une grosse *Instruction Window*, plusieurs instructions peuvent quitter cele-ci durant un seul cycle : il suffit que plusieurs ALU démarrent une nouvelle instruction.

Mais les *Reservations Stations* ont aussi des inconvénients. Elles sont souvent sous-utilisées et partiellement remplies. Il est rare que la totalité d'une *Reservation Station soit utilisé*, tandis qu'une *Instruction Window* n'a pas ce problème et est souvent remplie en totalité. Niveau rentabilité circuits/remplissage, l'*Instruction Window* est la grande gagnante.

Quelques détails

Ces *Instruction Window* sont composées d'entrées, des blocs de mémoires qui servent à stocker une instruction décodée. Lorsqu'une instruction est décodée, elle réserve une entrée rien que pour elle. Cette entrée est libérée dans deux cas : soit quand l'instruction est envoyée aux ALU, soit quand elle termine son exécution. Cela dépend du processeur et de son pipeline. Le rôle du *Scheduler* est de vérifier le contenu des entrées pour savoir quelle instruction *Issue*. Si plusieurs instructions sont prêtes à être issue sur la même ALU, ce dernier doit faire un choix. Et généralement cela signifie "issue la plus ancienne".

Il existe deux grands types d'*Instruction Window*. Ces deux types dépendent du contenu de celle-ci. Dans le premier type, l'*Instruction Window* ne stocke pas les opérandes de l'instruction : une entrée ne contient que notre instruction, et puis c'est tout. Le *Scheduler* sait quand les opérandes d'une instructions sont disponibles et quand il n'y a plus de dépendances grâce au *Scoreboard*. Dans le second type, l'*Instruction Window* contient aussi les opérandes d'une instruction, celles-ci étant envoyée à

l'*Instruction Window* par le réseau de Bypass. Les dépendances sont alors détectables sans aucun *Scoreboard*.

L'algorithme de Tomasulo et le renommage de registres

Dans le chapitre précédent, on a vu un premier algorithme d'exécution *Out Of Order* qui permettait de remplir certains vides dans notre pipeline. Mais celui-ci était toujours limité par certaines dépendances de données, qui avaient tendance à créer pas mal de vides, et contre lesquels notre *scoreboarding* ne pouvait pas faire grand chose. Pour limiter la casse, les concepteurs d'ordinateurs se sont creusés les méninges et ont fini par trouver une solution et ont réussit un exploit : virer la grosse majorité des dépendances de données lors de l'exécution.

Le renommage de registres

Dans ce que j'ai dit précédemment, j'ai évoqué trois types de dépendances de données. Certains d'entre elles sont des dépendances de données contre lesquelles on ne peut rien faire : ce sont de "vraies dépendances de données". Une vraie dépendance de donnée correspond au cas où une instruction a besoin du résultat d'une instruction précédente pour fonctionner : il s'agit des dépendances *Read After Write*. Ces dépendances sont aussi appelées des **dépendance de donnée véritables** ou une ***True Dependency***, ce qui montre bien leur caractère inévitable. Il est en effet impossible d'exécuter une instruction si les données nécessaires à son exécution ne sont pas encore connues. Quelque soit la situation, on est obligé de tenir compte de ces dépendances, qui imposent un certain ordre d'exécution de nos instructions, limitant ainsi les possibilités pour notre *Out Of Order*.

Des dépendances fictives

Évidemment, si on qualifie ces dépendances de dépendances véritables, c'est forcément que les autres sont un peu moins méchantes que prévu. Les dépendances, *Write After Write* et *Write After Read* sont en effet de fausses dépendances : elles viennent du fait que nos deux instructions doivent se partager le même registre ou la même adresse mémoire, sans pour autant que l'une aie besoin du résultat de l'autre. Ces dépendances viennent simplement du fait qu'une adresse mémoire (pardonnez mon abus de langage en confondant l'adresse et le *byte* qu'elle sélectionne), un registre, ou toute autre portion de la mémoire, est réutilisée pour stocker des données différentes à des instants différents. De telles dépendances sont appelées des ***name dependencies***. Et réutiliser une portion de la mémoire et remplacer une donnée devenue inutile par une autre ne pose pas de problèmes quand on exécute des instructions l'une après l'autre, mais ce n'est pas le cas quand on veut paralléliser ces instructions ou changer leur ordre.

Supprimer ces dépendances est donc une nécessité si on veut paralléliser nos instructions. Tirer parti des architectures *Out Of Order* nécessite de virer au maximum ces dépendances de données fictives afin de laisser un maximum d'opportunité pour éxecuter des instructions indépendantes en parallèle, ou pour changer l'ordre des instructions le plus facilement possible.

Mais comment faire ?

Ces dépendances apparaissent si on lit ou écrit des données différentes au même endroit : si ce n'est pas le cas, on peut trouver une solution ! Et cette solution est assez simple : on doit faire en sorte de ne pas réutiliser une portion de mémoire qui contient une donnée. C'est bien plus facile à dire qu'à faire, et rarement faisable : quand on est limité à 4 ou 8 registres sur un processeur, réutiliser au maximum les registres est au contraire une nécessité si on veut éviter de devoir accéder à la RAM ou au cache pour tout et n'importe quoi.

Bien évidemment, l'imagination débridée des concepteurs de processeur a déjà trouvée une solution à ce problème dans le cas de dépendances dues à un partage de registres entre deux instructions. Cette solution s'appelle le **renommage de registre**, et permet d'éliminer ces fausses dépendances de données *Write after Read* ou *Write after Write*.

Le renommage de registres

Prenons le cas d'une dépendance WAR : on a une lecture suivie par une écriture. Si on décide de changer l'ordre des deux accès mémoires et que l'on effectue l'écriture avant la lecture, la lecture ne renverra pas la valeur présente avant l'écriture, mais celle qui vient d'être écrite. Il est donc strictement impossible de changer l'ordre des lectures/écritures dans ce cas précis : si on le fait, le résultat enregistré en mémoire change et notre programme ne fait pas ce qu'il faut. Toutes les lectures de notre registre ou adresse mémoire précédant notre écriture devront être terminées avant de pouvoir lancer notre écriture.

Plus généralement, si on change l'ordre de deux instructions ayant une dépendance de donnée WAW ou WAR, nos deux instructions vont utiliser le même registre pour stocker des données différentes en même temps. Manque de chance, un registre ne peut contenir qu'une seule donnée, et l'une des deux sera perdue.

Du moins, c'est le cas si on lit ou écrit au même endroit. Si ce n'est pas le cas, on n'observe pas d'apparition de dépendances.Et une solution simple vient alors à l'esprit. Il suffit de conserver ces différentes versions du même registre dans plusieurs registres

séparés, et choisir le registre adéquat à l'utilisation. Ainsi, les opérations de lecture seront redirigées vers la copie de l'ancienne valeur (celle à lire), tandis que notre écriture écrira son résultat dans un autre registre. Une fois que toutes les lectures précédant notre écriture (dans l'ordre des instructions imposé par le programmeur) seront terminée, il suffira de remplacer l'ancienne valeur (la copie utilisée pour les lectures) par la nouvelle valeur (celle qui a été écrite).

L'idée est donc de stocker plusieurs versions d'un même registre dans des registres séparés. A chaque fois qu'on veut écrire ou modifier le contenu d'un registre, cette écriture est redirigée vers un autre registre. A chaque écriture, on utilise un nouveau registre pour stocker temporairement notre donnée à écrire. Et toutes les lectures vers cette version du registre sont redirigées vers ce registre supplémentaire. Une fois qu'une donnée, une version d'un registre, devient inutile, celui-ci devient réutilisable.

Tout ce qui peut être lu ou écrit peut subir ce genre de modification, mais en pratique, seuls les registres subissent ce genre de traitement. Certains processeurs permettent ainsi de faire la même chose sur des adresses mémoires, mais sont vraiment rares.

Des registres en double !

On voit bien qu'un seul registre architectural correspondra en réalité à plusieurs registres réellement présents dans notre processeur, chacun contenant le résultat d'une écriture. On a besoin non seulement de nos registres architecturaux, mais aussi de registres supplémentaires.
Ainsi, sur les processeurs implémentant le renommage de registres, il faut bien faire la distinction entre :

- les **registres architecturaux**, qui sont ceux définis par l'architecture extérieure du CPU, mais qui sont fictifs ;
- et les **registres physiques**, réellement présents dans notre CPU, bien plus nombreux que les registres architecturaux.

Le renommage de registres consiste à attribuer un ou plusieurs registres physiques pour chaque registre architectural lors de l'exécution d'un programme, en fonction des besoins, cette correspondance pouvant varier au cours du temps. Le renommage de registre permet ainsi de dupliquer chaque registre architectural en plusieurs exemplaires, chacun utilisant un registres physique.

Petite remarque : le nombre des registres physiques correspondant à un registre architectural variera dans le temps, suivant le nombre d'instructions qui doivent écrire dans un même registre.

Reprenons notre exemple de deux instructions devant utiliser le même registre. Le renommage de registres permet à deux registres physiques de devenir le même registre architectural : un de ces registres architecturaux sera utilisé par la première instruction, et l'autre par la seconde instruction. Les deux instructions n'utilisent plus les mêmes registres, et peuvent travailler chacune sur leur registre, indépendamment de l'autre : les dépendances de données disparaissent et les deux instructions peuvent alors s'exécuter en même temps.

C'est fait comment ?

Dans nos processeurs, les registres sont identifiés par ce qu'on appelle un **nom de registre**. Ce terme doit vous rappeler quelque chose, et ce n'est pas un hasard : on a vu cela dans le chapitre sur l'assembleur. Quoiqu'il en soit, nos registres sont donc identifiés par ces noms de registres qui ne sont autre que des suites de bits dont la valeur va permettre d'identifier un registre architectural parmi tous les autres. Nos registres physiques sont eux aussi identifiés par un nom de registre, mais qui est seulement connu du processeur : le ***Tag***. Pour attribuer un registre architectural à un registre physique, **il suffit de remplacer le nom du registre architectural par le *tag* du registre physique qui lui est attribué**. On dit alors que le registre architectural est renommé.

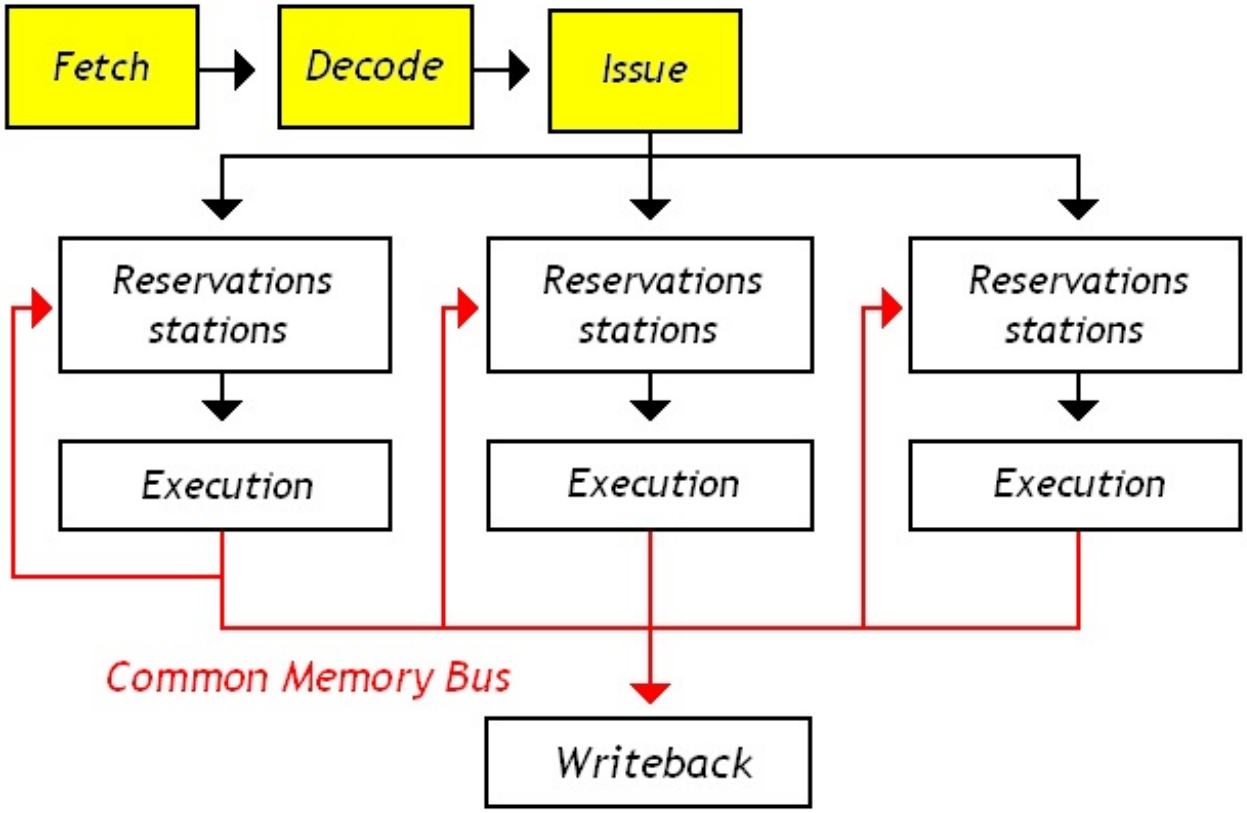
Ce remplacement est effectué dans un étage supplémentaire du pipeline. Celui-ci va prendre sur ses entrées le nom des registres architecturaux à manipuler, et va fournir sur sa sortie les *tag* des registres physiques correspondant.

Reservations stations

Le *Scoreboarding* est un algorithme assez simple. Mais il a un petit défaut : il ne gère pas le renommage de registres. Pour gérer le renommage de registre, il existe un autre algorithme d'exécution *Out Of Order*. Il s'agit de l'***algorithme de Tomasulo***. Cet algorithme a un autre avantage : il essaye d'utiliser du *forwarding* dès que possible afin de limiter la casse ! Voyons un peu comment est effectué le renommage de registre avec cet algorithme.

Aperçu

Pour commencer, voici un aperçu du pipeline d'un processeur utilisant cet algorithme, ci-dessous.



Nous verrons dans la suite à quoi ces étapes correspondent. Mais je tiens à préciser une chose : les étapes marquées en jaune sont des étapes qui ne sont pas concernés par l'*Out Of Order*. C'est à dire que les instructions sont chargées depuis la mémoire, décodées, et Issuées dans l'ordre du programme. C'est à la suite de ces étapes que les instructions pourront s'exécuter dans le désordre.

A première vue, rien n'a changé comparé au pipeline d'un processeur normal. Seul petit ajout : *Reservations Stations*. Tout l'algorithme se base sur les *Reservations Stations*. Ce sont elles qui servent de registres virtuels. Et cela aura des conséquences sur le reste du processeur. Par exemple, l'étage d'*Issue* sera maintenant en charge du renommage de registres. Pour comprendre le fonctionnement de cet algorithme, nous allons voir en détail chaque étape du pipeline, dans l'ordre.

Issue

L'étape d'*Issue* a deux utilités. Pour commencer, c'est elle qui devra renommer les registres. C'est elle qui décidera de donnée un *Tag* aux résultats de nos instructions, notamment. Ensuite, c'est elle qui vérifiera si on disposera des ressources nécessaires pour s'exécuter. Si ce n'est pas le cas, l'unité d'Issue va bloquer les étages précédents du pipeline, afin de faire patienter notre instruction à l'étage d'*Issue*.

On remarque tout de même une petite chose assez amusante avec cette unité d'*Issue*. Celle-ci va décider d'envoyer une instruction dans la suite du pipeline, mais elle ne va pas vérifier les dépendances. Après tout, on pourrait très bien imaginer une unité d'*Issue* qui vérifie si toutes les opérandes sont disponibles. Mais ici, ce n'est pas le cas. La gestion de l'*Issue* est séparée de la détection des dépendances. Cette séparation entre *Issue* et détection des dépendances s'appelle le ***Schelving***. Il existe d'autres techniques de *Schelving*, dont certaines se foutent du renommage de registres. Mais toutes ces techniques nécessitent de stocker les instructions entre l'étage d'*Issue* et celui de vérification des dépendances.

Ce stockage va se faire dans une sorte de mémoire tampon, qui stockera les instructions dont on doit s'occuper des dépendances. Tout ce qu'à à faire l'étape d'Issue, c'est renommer les registres et vérifier qu'il y a suffisamment de place dans cette mémoire tampon.

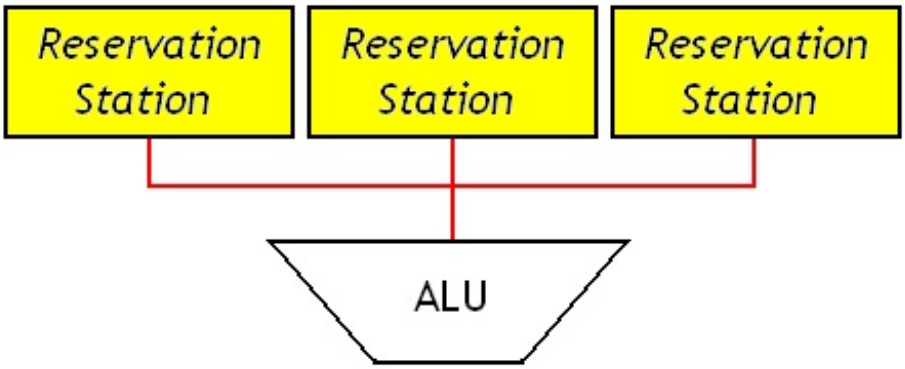
Reservation Stations

Mais voyons plus en détail le rôle de cette mémoire tampon. Une fois qu'une instruction a étè décodée et que ses registres ont

étés renommés, celle-ci est alors libre de toute dépendance WAR ou WAW. Le renommage de registre, effectué par l'étage d'*Issue*, a bien fait son travail. Mais rien n'indique que ses opérandes sont prêtes. Il se peut qu'une dépendance RAW vienne jouer les trouble-fête.

Pour résoudre cette dépendance RAW, l'instruction est mise en attente en attendant que ses opérandes soient disponibles. Jusque-là, pas grande différence avec le *scoreboarding*. Mais la nouveauté vient de la façon dont cette instruction est mise en attente. Avec le *Scoreboarding*, notre instruction était mise en attente, mais ses opérandes étaient lues depuis les registres architecturaux, une fois disponibles. On ne pouvait pas lire depuis un registre virtuel.

Avec l'algorithme de Tomasulo, notre instruction sera stockée dans une grosse mémoire tampon, située juste avant les unités de calcul. Chaque unité de calcul se voit attribuer une de ces mémoires tampon, qui lui est propre et à laquelle elle seule peut accéder. Cette mémoire tampon va servir à stocker aussi les signaux de commande de nos instructions en attente, ainsi que leurs opérandes. Pour cela, notre mémoire tampon est composée de blocs bien organisés. Chacun de ces blocs s'appelle une ***Reservation Station***.

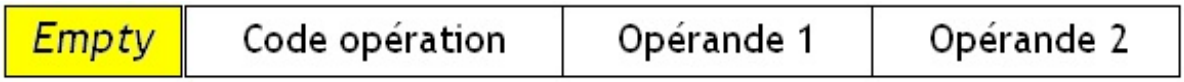


Chaque *Reservation Station* peut stocker toutes les informations nécessaires pour éxecuter une instruction. C'est à dire ses signaux de commande, et ses opérandes.

Empty Bit

Lorsque notre unité d'*Issue* va vouloir éxecuter une instruction, elle va d'abord devoir vérifier si une *Reservation Station* est libre devant l'unité de calcul devant éxecuter notre instruction. Si ce n'est pas le cas, l'unité d'*Issue* va devoir se bloquer, et bloquer aussi les étages du pipeline qui la précédent. Dans le cas contraire, si une *Reservation Station* est libre, l'unité d'Issue va la remplir avec les informations nécessaires sur son exécution.

Reste à savoir comment fait notre unité d'*Issue* pour savoir que nos *Reservation Stations* sont libres. Pour cela, chaque *Reservation Station* possède un bit nommé *Empty* qui indique si celle-ci est vide ou remplie par une instruction. Quand une *Reservation Station* se remplie, elle positionne automatiquement ce bit à 1. L'unité d'*Issue* a juste à lire des bits des différentes *Reservation Station* pour savoir lesquelles sont libres.

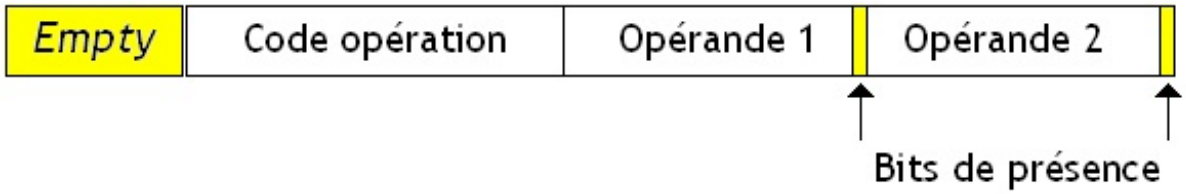


Valid Bit

Une instruction placée dans une *Reservation Station* sera exécutée une fois que toute ses opérandes seront disponible (sauf optimisations bizarres). Pour éxecuter l'instruction au bon moment, il faut trouver un moyen pour savoir que toutes les opérandes d'une instruction sont disponibles.

Pour cela, il suffit d'ajouter des bits de présence qui permettront de savoir si l'opérande est bien écrite dans la *Reservation*

Station. Quand une opérande est écrite dans la *Reservation station*, le bit de présence correspondant sera mis à jour. Des circuits reliés à la *Reservation station* se chargeront alors de vérifier à chaque cycle d'horloge si toutes les opérandes de l'instruction sont disponible.



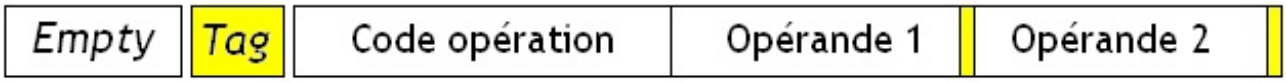
On remarque facilement une chose : toutes les instructions n'ont pas le même nombre d'opérandes. Et donc, ces circuits doivent décider quand démarrer une instruction en fonction non seulement des opérandes disponibles dans la *Reservation Station*, mais aussi de l'instruction : on doit donc utiliser l'opcode de l'instruction pour cette décision.

Tag

Je ne sais pas si vous avez remarqués, mais ces *Reservation Station* sont des registres virtuels. Ces sont elles qui stockent les opérandes de nos instructions. Au lieu de lire des données depuis le *Register File*, non données peuvent être lues directement depuis la sortie de l'ALU. Reprenez le schéma décrivant l'algorithme de Tomasulo, que j'ai mit au-dessus. Vous verrez qu'il existe une sorte de bus qui relie les sorties des ALUs directement sur leurs entrées. Et bien sachez que ce mécanisme permet non seulement de faire du Bypass de façon efficace, mais qu'il permet aussi de faire du renommage de registre une fois couplé aux *Reservation Station*.

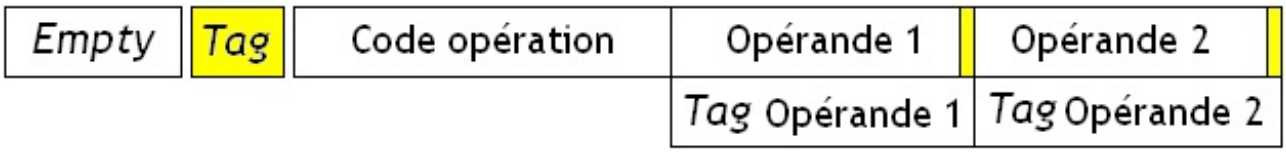
En fait, nos *Reservation Station* vont stocker des données directement sorties de l'ALU, et qui ne sont pas forcément enregistrées dans le *Register File*. Sans *Reservation Station*, on devrait attendre que le registre soit libre, c'est à dire attendre que toutes les lectures de ce registre soient terminées. A la place, on préfère écrire dans une *Reservation Station*, qui stockera le résultat temporairement, avant qu'il soit recopié dans le *Register File*. On voit bien que nos *Reservation Stations* servent de registres virtuels.

Pour gérer le processus de renommage, une instruction va devoir indiquer dans quels registres virtuels stocker son résultat. Pour cela, chaque instruction se voit attribuer un *Tag*, qui permet d'identifier ce résultat parmi tous les autres. Ce Tag est stocké directement dans la *Reservation Station*, à coté des opérandes, de l'opcode de l'instruction et de tout le reste.



Opérandes Sources

Ce résultat, une fois fourni par l'unité de calcul, peut être réutilisé par une autre instruction. Si c'est le cas, il doit être stocké dans la *Reservation Station* attribuée à l'instruction ayant besoin de ce résultat. Pour identifier celle-ci, on utilise le fait que chaque *Reservation Station* stocke les opérandes de l'instruction qu'elle stocke. Chacune de ces opérandes est fournie par une instruction : il s'agit donc d'un résultat d'instruction comme un autre, qui est fourni par une unité de calcul. Et ce résultat possède un *Tag*. Pour chaque opérande stockée dans une *Reservation Station*, il faut donc ajouter de quoi stocker les *Tags* de cette opérande.



Lorsqu'une instruction renvoie un résultat, celui-ci est envoyé à toutes les *Reservation Station* via le *Common Memory Bus*. Celles-ci vont alors comparer chacun des Tags de leurs opérandes avec celui-ci de l'opérande qui est disponible sur ce bus. Si il y a correspondance, le résultat est alors stocké dans la *Reservation Station*.

Dispatch

Une fois passée l'étape d'*Issue*, les instructions doivent attendre que leurs opérandes soient disponibles dans les *Reservations Stations*. Elles sont d'abord *Issue*, puis elles sont autorisées à s'exécuter. Cette autorisation est une étape particulière de

l'algorithme de Tomasulo. Il s'agit du ***Dispatch***.

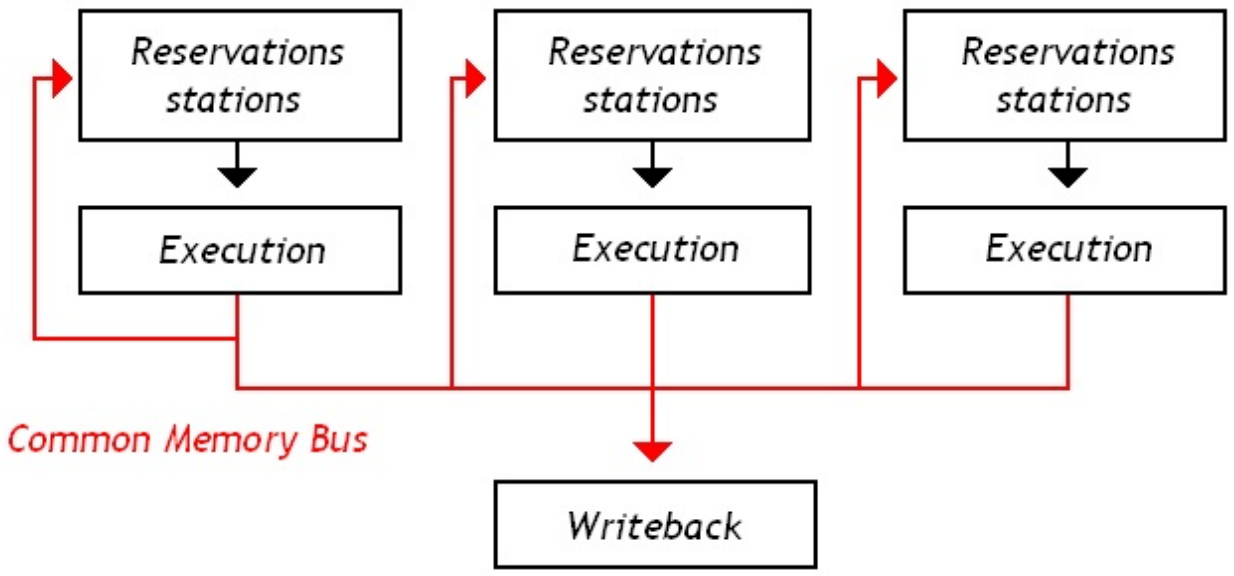
Dans le cas le plus simple, les instructions sont dispatchées dans leur ordre d'arrivée. Mais pour plus d'efficacité, il est préférable de dispatcher des instructions dès que leurs opérandes sont prêtes. Dans ce cas, l'étape de *Dispatch* est gérée par un circuit qui regarde la disponibilité des opérandes, et qui décide quelle instruction envoyer à l'ALU. Il faut bien sûr traiter le cas où plusieurs instructions ont leur opérandes prêtes en même temps. Dans ce cas, c'est souvent l'instruction la plus ancienne qui passe en premier.

Le *Common Memory Bus*

Notre instruction a fini de s'exécuter. Son résultat est enfin connu, et il va alors être distribué à tous les circuits qui en ont besoin. C'est l'étape de ***Completion***. Cette complétion se fait dans le désordre le plus total : dès qu'une instruction fournit un résultat, elle l'envoie vers les unités chargés d'écrire dans les *Reservations Stations*, dans les registres, et dans les unités chargées de communiquer avec la mémoire.

Bypassing

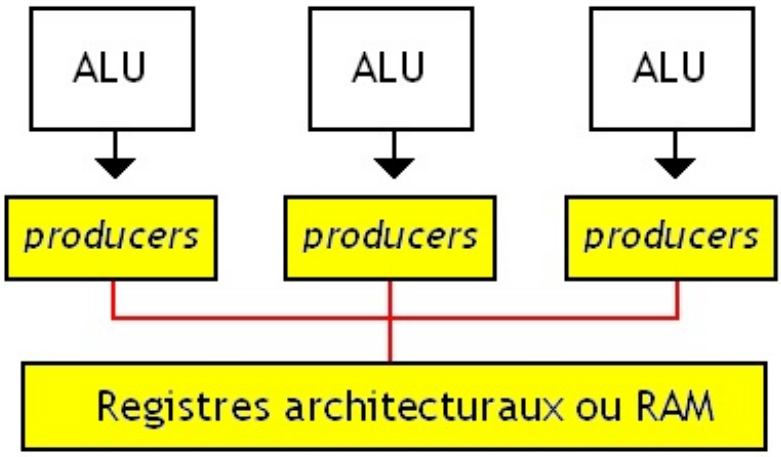
Avec l'algorithme de Tomasulo, on peut renvoyer ce résultat dans une *Reservation Station* et ainsi effectuer du *Bypassing*. Pour cela, le résultat va être envoyé sur un bus qui distribuera le résultat calculé aux circuits de *Writeback* ainsi qu'aux *Reservations Stations*. Ce bus s'appelle le ***Common Memory Bus***.



Ce résultat est envoyé sur le *Common Memory Bus* avec le Tag de l'instruction qui l'a calculé.

Producers

Pour éviter les problèmes sur les processeurs utilisant plusieurs unités de calculs, nos unités de calcul ne vont pas écrire directement sur ce bus : imaginez les problèmes d'arbitrage que cela pourrait donner ! A la place, on va en intercaler des registres entre les unités de calcul et ce bus. Ces registres sont appelés les ***Producers***. Seul un de ces *Producers* pourra transférer ces données dans le *Common Memory Bus* à un moment donné : si plusieurs *Procuders* veulent écrire sur le bus, seul un seul d'entre eux sera autorisé à le faire, ce qui élimine le problème évoqué au-dessus.



Quand celui-ci est libre, un de ces *Producers* sera choisit et pourra envoyer son contenu dans le *Common Memory Bus*. En choisissant correctement le *Producer* à chaque fois, on est certain que les données seront envoyées dans le bon ordre.

Accès mémoires

Il faut préciser une chose : les accès à la mémoire principale (la RAM) sont traités à part des autres instructions. En effet, ces instructions d'accès mémoires doivent s'exécuter dans l'ordre imposé par le programme. On ne sait pas supprimer les dépendances WAR et WAW sur ces instructions : le renommage de registre ne permet de supprimer de telles dépendances que pour les registres. Pour éviter les problèmes, notre processeur contient des unités spécialisés, qui prennent en charge les instructions de lecture (`Load`) ou d'écriture (`Store`) en mémoire. Ce sont les ***Load Unit*** (pour les lectures) et les ***Store Unit*** (pour les écritures).

Ces unités sont reliées au *Common Memory Bus*. Lorsqu'une donnée est lue depuis la mémoire, elle atterrit aussi bien dans le *Register File* que Dans les *Reservations Stations*. Petite remarque : l'unité chargée des lecture est aussi précédée par des *Reservations Stations* qui lui sont réservées. Elles servent à mettre en attente des instructions de lecture dont on ne connait pas encore l'adresse. L'instruction est alors stockée, et démarre quand la mémoire est libre, et que l'adresse à lire est connue.

Pour les écritures, c'est un peu différent. Si jamais une donnée doit être écrite, celle-ci est envoyée sur le *Common Memory Bus* et réceptionnée par l'unité d'écriture. Celle-ci sera alors stockée dans une *Reservation Station*, placée juste devant l'unité dédiées aux écritures. Une fois que la donnée et l'adresse à laquelle écrire sont connues, et que la mémoire est libre, l'écriture est lancée.

Bilan

On le voit, cet algorithme se base sur l'existence de *Reservations Stations* pour effectuer du renommage de registres. Dans la version vue plus haut de l'algorithme de Tomasulo, ces *Reservations Stations* servent de registres virtuels. On verra dans la suite que ce n'est pas toujours le cas.

L'exécution d'une instruction avec cet algorithme de Tomasulo est donc découpée en plusieurs étapes. Ce découpage en étape peut servir à organiser le pipeline d'un processeur.
Aussi, il n'est pas rare que les processeurs *Out Of Order* utilisant le renommage de registres aient un pipeline similaire à celui-ci :

- **Fetch** ;
- **Decode** ;
- **Issue/Rename** : renommage des registres et allocation dans les *Reservations Stations* ;
- **Dispatch** : autorisation d'exécution, parfois précédée d'un temps d'attente des opérandes dans les *Reservation Stations* ;
- **Execution** ;
- **Completion** : le résultat est disponible sur le *Common Memory Bus* et enregistré dans les registres.

Mais pour le moment, il y a une chose que l'on a oublié de dire. Si nos *Reservations Stations* sont des registres virtuels, les écritures dans le *Register File* se font à l'étape de *Completion*. C'est lors de cette étape que les écritures en mémoire et dans les registres s'effectuent. Et cette *completion* se fait dans le désordre le plus total.

Re-Orders Buffers

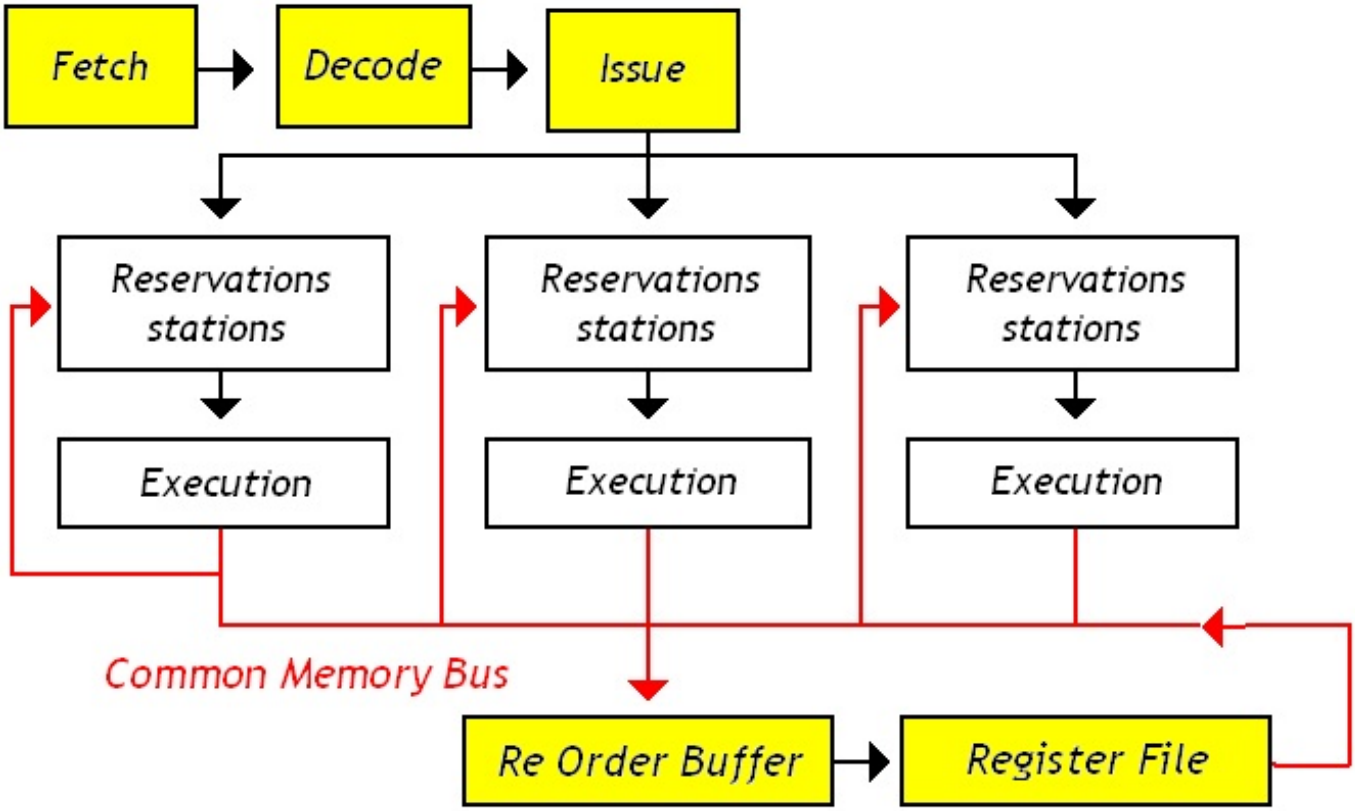
La version de l'algorithme de Tomasulo que l'on a vue a un léger problème : la moindre interruption ou mauvaise prédiction de branchement peut tout casser ! Les mauvaises prédiction de branchements, les interruptions et les exceptions matérielles peuvent mettre un sacré bazar si l'ordre des lectures/écritures n'est pas celui prévu par le programmeur. Si une instruction qui n'aurait pas du être exécutée (parce que précédée par un branchement, une interruption, ou une exception dans l'ordre du

programme) modifie un registre alors qu'elle n'aurait pas du, on est mal. Pour éviter ce genre de problèmes et gérer les interruptions et branchements correctement, notre processeur va se charger de "remettre les lectures et écritures dans l'ordre".

Pour cela, les résultats des instructions vont devoir attendre avant d'être enregistrés dans les registres ou la mémoire. Cette attente dure jusqu'à ce que toutes les instructions qui précédent l'instruction qui a fournie le résultat (dans l'ordre normal du programme) soient toutes exécutées.

Cette attente se fait dans un étage supplémentaire du pipeline, rajouté à la suite des précédents. Un nouvel étage fait son apparition : **Commit**.
Le pipeline de notre processeur devient donc :

- **Fetch** ;
- **Decode** ;
- **Issue/Rename** : renommage des registres et allocation dans les Reservations Stations ;
- **Execution** ;
- **Completion** : le résultat est disponible sur le *Common Memory Bus* et mis en attente ;
- **Commit** : le résultat est enregistré dans les registres.



Le *Reorder Buffer*

Pour s'assurer que les résultats sont écrits dans l'ordre du programme, et pas n'importe comment, notre *Reorder Buffer* est organisé d'une certaine façon.

Ce *Reorder Buffer* est composé de plusieurs entrées, des espèces de blocs dans lesquels il va stocker des informations sur les résultats à écrire, et leur ordre. Ainsi, il saura prendre en charge les écritures dans les registres, et les remettre dans le bon ordre. Voyons un peu son contenu.

Tag

Il va de soit que notre *Reorder Buffer* contient les données à écrire. Mais il doit aussi les remettre dans l'ordre.

Lorsqu'une instruction vient d'être décodée durant l'étape d'*Issue*, celle-ci est alors ajoutée dans le *Reorder Buffer* à la suite des autres. Les instructions étant décodées dans l'ordre du programme, l'ajout des instructions dans le *Reorder Buffer* se fera dans l'ordre du programme, automatiquement. Elle va réserver une entrée dans le *Reorder Buffer*. Pour savoir à quelle instruction cette entrée est réservée, on y ajoute le *Tag* de l'instruction.

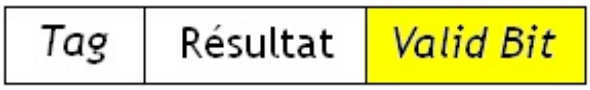
En conséquence, une entrée de notre *Reorder Buffer* contient donc de quoi stocker un résultat, et aussi de quoi placer le *Tag* d'une instruction.

Lorsque le résultat de l'instruction est connu, il est disponible sur le Common Memory Bus avec son *Tag*. A chaque envoi sur ce bus, le *Reorder Buffer* va regarder s'il y a une correspondance entre ce *Tag*, et un des *Tags* qu'il contient. Si c'est le cas, le résultat est écrit dans le *Reorder Buffer*, dans l'entrée dont le *Tag* correspond.

Valid Bit

Maintenant, prenons un cas particulier. Imaginons que nous ayons une instruction dans notre *Reorder Buffer*. Toutes les instructions précédentes ont écrit leurs résultats en mémoire ou dans les registres. Je suis sur que vous pensez que l'on peut écrire le résultat de notre instruction dans les registres sans problème. Elle peut alors quitter le *Reorder Buffer*.

Mais et si son résultat n'est pas disponible ? Il faut alors garder cette instruction dans le *Reorder Buffer*. Pour savoir si notre instruction a bien produit un résultat ou s'il faut attendre, chaque entrée du *Reorder Buffer* contient un **bit de presence**. Celui-ci est mit à 1 quand le résultat de l'instruction est écrit dans l'entrée.



Autres informations

Suivant le processeur, le *Reorder Buffer* peut parfois contenir d'autres informations, comme le registre ou l'adresse où écrire. Mais cela dépend du processeur utilisé, aussi cela ne sert à rien de rentrer dans les détails.

Une File

Il faut préciser que les instructions ne quittent pas le *Reorder Buffer* n'importe quand. Un résultat est enregistré dans un registre lorsque les instructions précédentes (dans l'ordre du programme) ont toutes écrit leurs résultats dans les registres. Pour cela, notre *Reorder Buffer* est une sorte de mémoire, triée dans l'ordre d'ajout : les instructions sont triées des plus récentes aux plus anciennes. Seule l'instruction la plus ancienne peut quitter le *Reorder Buffer* et enregistrer son résultat une fois si celui-ci est présent (pensez au bit de présence). Les instructions plus récentes doivent attendre.

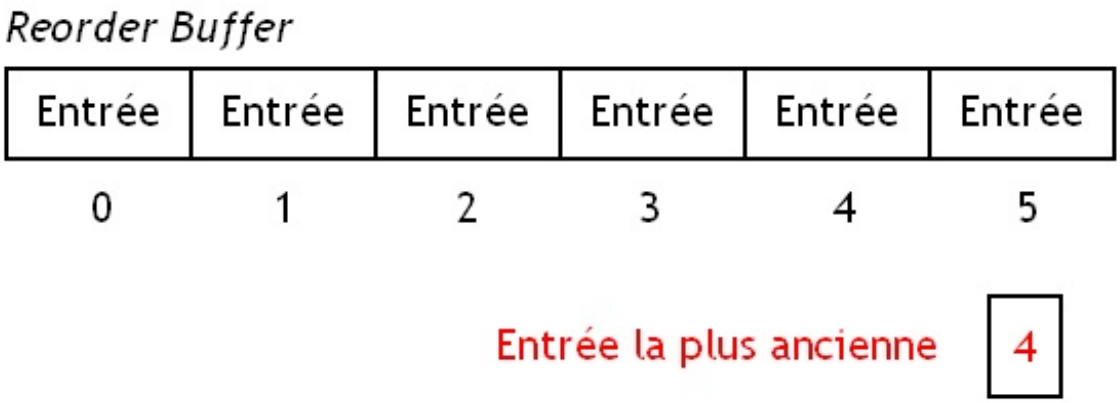
Implémentation

Voyons un peu comment gérer cet ordre d'écriture. Notre *Reorder Buffer* contient un nombre d'entrée qui est fixé, câblé une fois pour toute. Chacune de ces entrées est identifiée par un nombre, qui lui est attribué définitivement.

Cet identifiant pourra alors servir à configurer un multiplexeur, qui sélectionnera la bonne entrée dans le *Reorder Buffer*, pour lire son contenu ou y écrire.

End Of Queue

Ces entrées sont gérées par leur identifiant. Le numéro de l'entrée la plus ancienne est ainsi mémorisé dans une petite mémoire. Cela permet de pointer sur cet entrée, qui contient la prochaine donnée à enregistrer dans les registres ou la mémoire.



Quand cette entrée quitte le *Reorder Buffer*, le numéro, le pointeurs sur la dernière entrée est augmenté de 1, pour pointer sur la prochaine entrée.

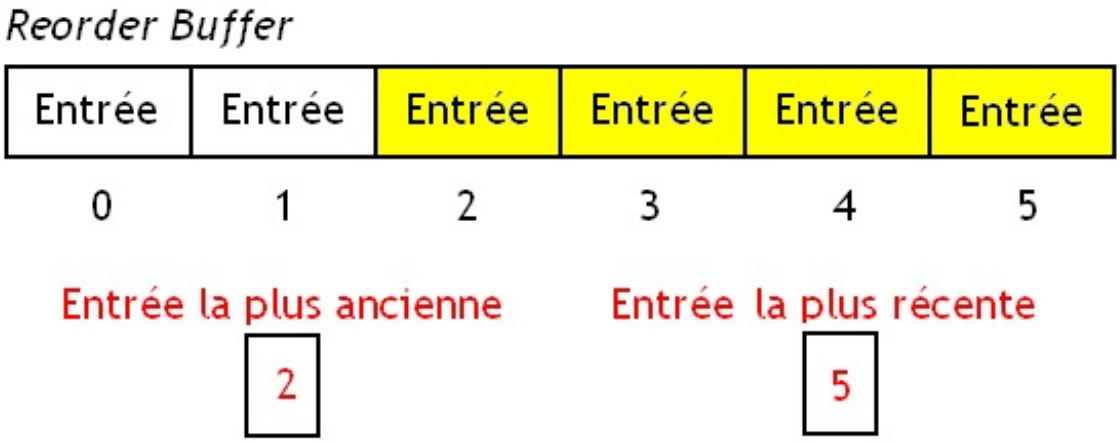
Begining Of Queue

De même, le numéro de l'entrée la plus récente est aussi mémorisé dans une petite mémoire. Il faut bien savoir où ajouter de nouvelles entrées. Ainsi, le *Reorder Buffer* sait quelles sont les entrées valides : ce sont celles qui sont situées entre ces deux entrées.

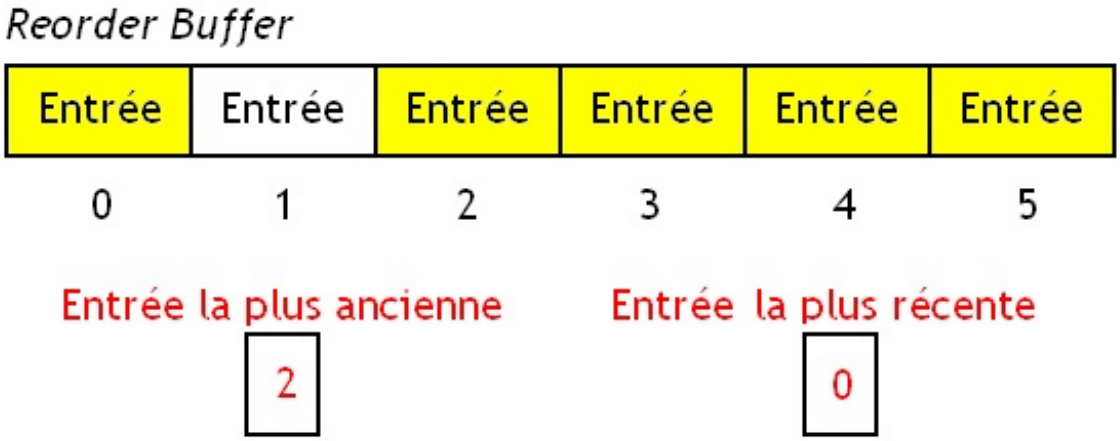
Quand on ajoute des instructions dans le *Reorder Buffer*, il ne faut pas oublier d'augmenter ce numéro de 1.

Petit détail : quand on ajoute des instructions dans le *Reorder Buffer*, il se peut que l'on arrive au bout, à l'entrée qui a le plus grand nombre. Pourtant, le *Reorder Buffer* n'est pas plein. De la place se libère dans les entrées basses, au fur et à mesure que le *Reorder Buffer*. Dans ce cas, on n'hésite pas à reprendre depuis le début.

Exemple : je prends l'exemple précédent, avec l'entrée 4 occupée. Si je rajoute une instruction, je remplirais l'entrée 5.



La prochaine entrée à être remplie sera l'entrée numéroté 0. Et on poursuivra ainsi de suite.



Taille du R.O.B

Quand notre *Reorder Buffer* est plein, le processeur va automatiquement bloquer les étages de *Fetch*, Décodage, et renommage (*Issue*). Cela évite de charger des instructions dans celui-ci alors qu'il est plein. Conséquence : cela empêche de charger de nouvelles instructions dans les *Reservations Stations*. Petite remarque : cela implique que plus un *Reorder Buffer* peu contenir d'instruction, plus on pourra éxecuter d'instructions de façon anticipée. L'exécution *Out-Of-Order* sera ainsi plus facile et donnera de meilleurs résultats.

Sur les processeurs utilisant un séquenceur micro-codé, certaines améliorations au niveau de l'unité de décodage d'instruction permettent de mieux utiliser ce *Reorder Buffer*. Ces fameuses améliorations sont celles qui consistent à fusionner une ou plusieurs instructions machines en une seule micro-opération. Notre *Reorder Buffer* va en effet stocker des micro-instruction, et non des instructions machines. En fusionnant plusieurs instructions machines ou micro-opérations en une seule, on diminue le nombre d'instructions à stocker dans le *Reorder Buffer* : on gagne un peu de place.

Spéculation Recovery

Dans le chapitre parlant de la prédiction de branchement, j'avais dit qu'il y avait deux solutions pour remettre le pipeline à son état originel en cas de mauvaise prédiction de branchement : empêcher nos instructions fautives d'enregistrer leurs résultats dans les registres (sous entendu, les registres architecturaux), ou remettre les registres à leurs valeurs originelles à partir d'une sauvegarde de ceux-ci. Le *Reorder Buffer* permet d'implémenter facilement la première solution. C'est lui qui empêche les instructions exécutées de façon anticipées d'enregistrer leurs résultats dans les registres architecturaux. en cas de mauvaise prédiction de branchement. Si une interruption ou une mauvaise prédiction de branchement a lieu, le *Reorder Buffer* se débarrassera des résultats des instructions qui suivent l'instruction fautive (celle qui a déclenché l'interruption ou la mauvaise prédiction de branchement) : ces résultats ne seront pas enregistrés dans les registres architecturaux. Les résultats des instructions précédant l'instruction fautive auront alors déjà étè sauvegardés, laissant le processeur propre.

Mais on doit améliorer le *Reorder Buffer* pour qu'il puisse gérer tout cela.

Où reprendre ?

Première chose, dans de telles situations, il faut savoir où reprendre. Pour cela, chaque entrée dans le *Reorder Buffer* contient en plus l'adresse de l'instruction. Par adresse, on veut dire contenu du *Program Counter* quand l'instruction a étè Fetchée. Le *Reorder Buffer* utilise cette adresse pour savoir à partir de quelle instruction il faut tout ré-exécuter. Parfois, cette adresse remplace le *Tag* de l'instruction.

Spéculative Bit

Pour cela, on rajoute un bit dans chaque entrée du *Reorder Buffer*. Ce bit servira à préciser si l'instruction a levée une exception ou s'il s'agit d'un branchement mal pris. Si ce bit est à 1, alors il y a eu une mauvaise prédiction de branchement ou une exception. Les pointeurs sur l'entrée la plus récente et la plus ancienne sont alors réinitialisés, mis à à zéro. Cela permet de vider le *Reorder Buffer*, et évite d'enregistrer des résultats faux en mémoire.

Sur certains processeurs, on fait une distinction entre branchements et exception, en utilisant un bit pour préciser une mauvaise prédiction de branchement, et un autre pour préciser l'exception (et si possible quelques autres pour préciser de quelle exception

il s'agit).

Champ type de l'instruction

Néanmoins, il faut prendre attention à un détail : certaines instructions ne renvoient pas de résultats. Sur certains processeurs, c'est le cas des branchements. On pourrait penser que ceux-ci ont pour résultat leur adresse de destination, mais ce n'est pas forcément le cas sur tous les processeurs. Tout dépend de comment ils gèrent leurs branchements. Mais on peut aussi citer le cas des lectures, qui ne fournissent pas de résultats.

La logique voudrait que vu que ces instructions n'ont pas de résultats, on ne doive pas leur allouer d'entrée dans le *Reorder Buffer*. Mais si on fait cela, nos exceptions et nos mauvaise prédictions de branchement risquent de ne pas reprendre au bon endroit. N'oubliez pas qu'on détermine à quelle adresse reprendre en se basant sur le *Program Counter* de l'instruction qui quitte le *Reorder Buffer*. Si on détecte une exception, c'est celui-ci qui est utilisé. Et si on ne place pas une instruction dans le *Reorder Buffer*, et que celle-ci lève une exception matérielle, on ne reprend pas au bon endroit.

Pour éviter cela, on ajoute quand même ces instructions dans le *Reorder Buffer*, et on rajoute un champ dans l'entrée. Ce champ stockera le type de l'instruction, afin que le *Reorder Buffer* puisse savoir s'il s'agit d'une instruction qui a un résultat ou pas.

On peut aussi utiliser cette indication pour savoir si le résultat doit être stocké dans un registre ou dans la mémoire. Sur certains processeurs, le *Reorder Buffer* s'occupe de toutes les lectures et écritures, qu'elles se fassent dans la mémoire et dans les registres. Et le *Reorder Buffer* doit trouver un moyen pour savoir s'il faut aller manipuler la RAM ou le *Register File*. De plus, dans certains *Reorder Buffer* stockent des informations supplémentaires, comme le registre ou l'adresse destination, dans des blocs de l'entrée. Cette indication sur le type de l'instruction permet ainsi d'indiquer au *Reorder Buffer* s'il doit utiliser le morceau de l'entrée qui stocke une adresse ou celle qui contient un registre de destination.

Accès mémoire

Pour remettre les accès mémoire dans l'ordre, certains processeurs utilisent un *Reorder Buffer* dédié aux lectures/écritures. Celui-ci est séparé du *Reorder Buffer* dédié aux registres. Mais les deux communiquent ensemble. Sur d'autres processeurs, on utilise un seul *Reorder Buffer*, chargé de traiter à la fois les instruction qui touchent aux registres et celles qui vont manipuler la mémoire.

Autres formes de renommages

Dans ce qu'on a vu dans l'algorithme de Tomasulo, on a vu l'importance du *Reorder Buffer*, ainsi que celle des *Reservation Station*. Ce sont ces dernières qui servent de " registres virtuels ", et qui permettent de faire du renommage de registre. Il ne s'agit pas à proprement parler de registres, mais les *Reservation Stations* ont exactement la même utilité que les registres virtuels. Le renommage de registres est alors implicite, dans le sens où il n'y a pas vraiment de registres virtuels, mais que quelque chose sert de registres virtuel. Mais il n'y a pas beaucoup de différences en terme d'efficacité avec d'autres formes de renommage de registre.

Il existe d'autres façons de faire du renommage de registre. La plupart sont des adaptations de l'algorithme de Tomasulo, adaptées afin d'être plus efficaces, ou moins consommatrices en énergie. Dans ces nouvelles version de l'algorithme de Tomasulo, les *Reservation Station* ne servent plus de registres virtuels, ces derniers étant stockés ailleurs. Cela peut paraitre bizarre, mais on va voir qu'en réalité, c'est plutôt intuitif.

ROB

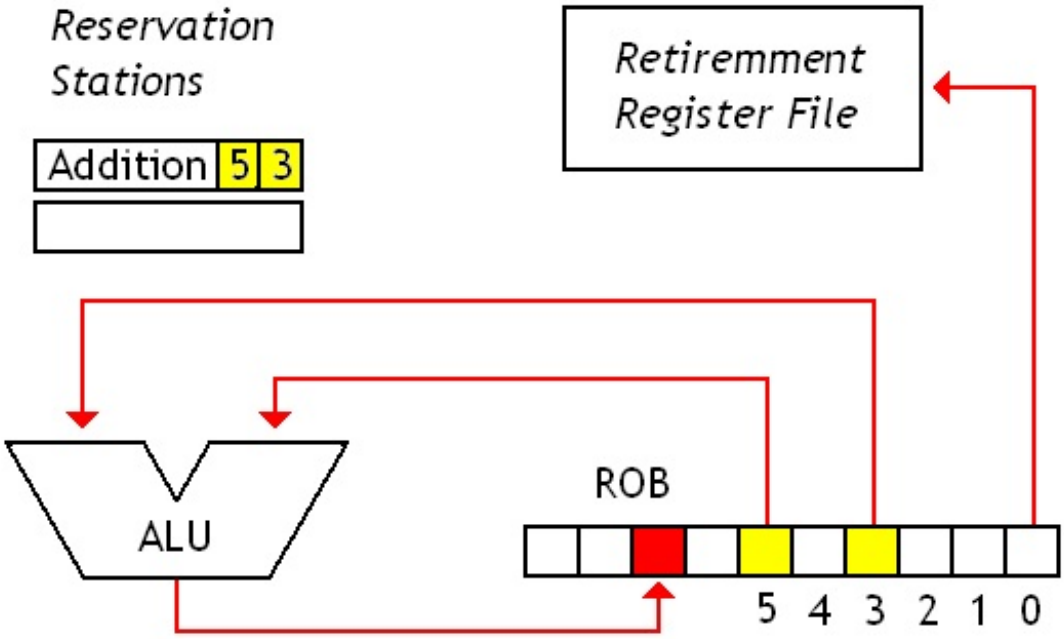
Commençons par faire une petite remarque sur l'algorithme de Tomasulo, auquel on a jouté un *Reorder Buffer*. Dans cette version de l'algorithme, les résultats de nos instructions sont stockés en deux exemplaires : dans les *Reservation Stations*, et dans le *Reorder Buffer*. Certains chercheurs ont alors eu l'idée de faire en sorte de ne pas copier inutilement nos données. Au lieu de placer nos données en double ou triple, ils ont décidés de voir si on ne pouvait pas changer l'algorithme pour n'avoir qu'un seul exemplaire de données.

L'idée est de stocker toutes les données en un seul exemplaire, dans le *Reorder Buffer*. On se retrouve donc avec un *Reorder buffer*, qui est équivalent aux registres virtuels, et un *Register File* qui contient tous les registres architecturaux. Avec éventuellement les fameuses *Reservations Stations*, mais sans que cela soit obligatoire : on peut très bien s'en passer et les remplacer par autre chose.

Fonctionnement

Avec cette technique, les *Reservation Stations* ne contiennent plus de données, mais vont stocker de quoi savoir dans quelle entrée du *Reorder Buffer* trouver la donnée. Les Tags des opérandes seront remplacés par le numéro des entrées du *Reorder Buffer* qui contiendra la donnée. Même chose pour le Tag du résultat d'une instruction, qui remplacé par le numéro de l'entrée du *Reorder Buffer* destinée à l'accueillir.

Quand une instruction a toutes ses opérandes de prêtes, celles-ci ne seront pas dans les *Reservation Station*, mais elles seront disponibles dans le *Reorder Buffer* ou dans le *Register File*. Notre instruction aura juste à les lire depuis le *Reorder Buffer* ou le *Register File*, et les charger sur les entrées de l'unité de calcul. Cela nécessite de connecter notre *Reorder Buffer* sur les entrées des unités de calcul, mais le jeu en vaut la chandelle.



Attention : ne prenez pas ce schéma trop à la lettre. Dans ce schéma, on pourrait croire que le *Register File* ne sert à rien. En fait, il faut se souvenir que les opérandes d'une instruction peuvent aussi être lues depuis le *Register File* : il n'y a pas que le *Reorder Buffer* qui peut contenir les opérandes.

De plus, je n'ai pas indiqué le *Common Memory Bus*, pour ne pas surcharger le tout. Et ce sera aussi le cas dans tous les schémas qui vont suivre.

De même, il arrive que certains processeurs fusionnent le *Reorder Buffer* et les *Reservation Stations* dans un seul gros circuit. On n'est vraiment pas à un détail prêt !

Pipeline

Une petite remarque : cette lecture des opérandes depuis un registre peut prendre un certain temps. Pour éviter tout problème, cette lecture est effectuée dans un étage du pipeline séparé.
Le pipeline de notre processeur devient donc :

- **Fetch** ;
- **Decode** ;
- **Issue/Rename** : renommage des registres et allocation dans les Reservations Stations ;
- **Reorder Buffer Read** : lecture des opérandes d'une instruction depuis le *Reorder Buffer* ;
- **Execution** ;
- **Completion** : le résultat est disponible sur le *Common Memory Bus* et mis en attente ;
- **Commit** : le résultat est enregistré dans les registres.

Pour information, cette version de l'algorithme de Tomasulo était utilisée dans d'anciens processeurs commerciaux, comme les Pentium Pro, le Pentium II, ou encore le Pentium III.

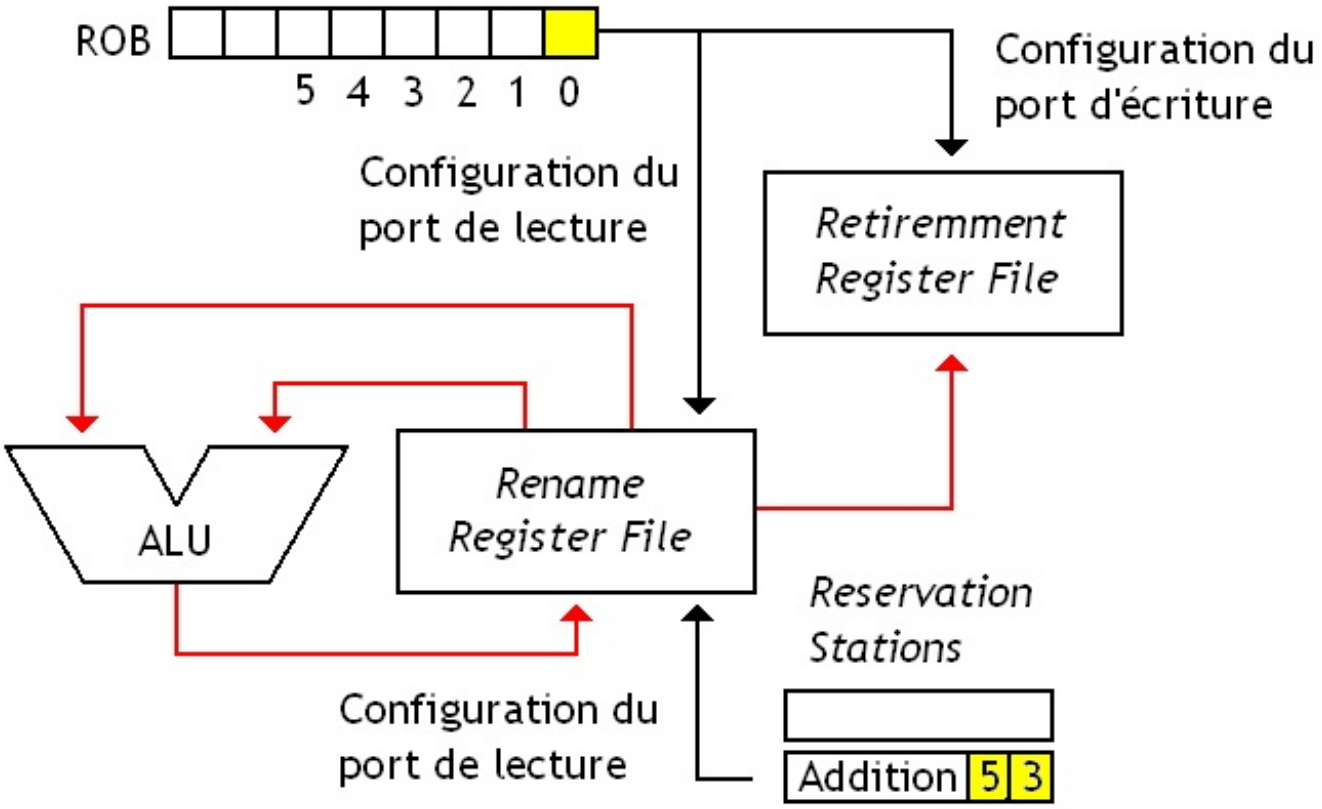
Rename Register File

Avec les deux versions précédentes de l'algorithme de Tomasulo, le renommage de registre était implicite : il n'y avait pas de *Register File* spécialisé, qui contenait les registres virtuels. A la place, on utilisait les *Reservation Station* ou le *Reorder Buffer*. Mais sur la majorité des processeurs actuels, le renommage de registre est explicite.

Cette fois-ci, on se retrouve avec un peu plus de composants. On dispose d'un *Register File* qui stocke les registres architecturaux, un autre *Register File* qui contient les registres virtuels, un *Reorder Buffer*, et éventuellement des *Reservations*

Stations. Le *Register File* qui stocke les registres virtuels porte un nom : c'est le *Rename Register File*. Evidemmemnt, le *Register File* qui stocke les registres architecturaux porte aussi un nom : c'est le *Retiremment Register File*.

Cette fois-ci, le *Reorder Buffer* subit le même traitement qu'on a fait subir aux *Reservations Stations* : il ne contient plus la moindre donnée. Avec cette version de l'algorithme, *Reorder Buffer* et *Reservations Stations* contiennent à la place le nom du registre virtuel ou réel qui contient la donnée.

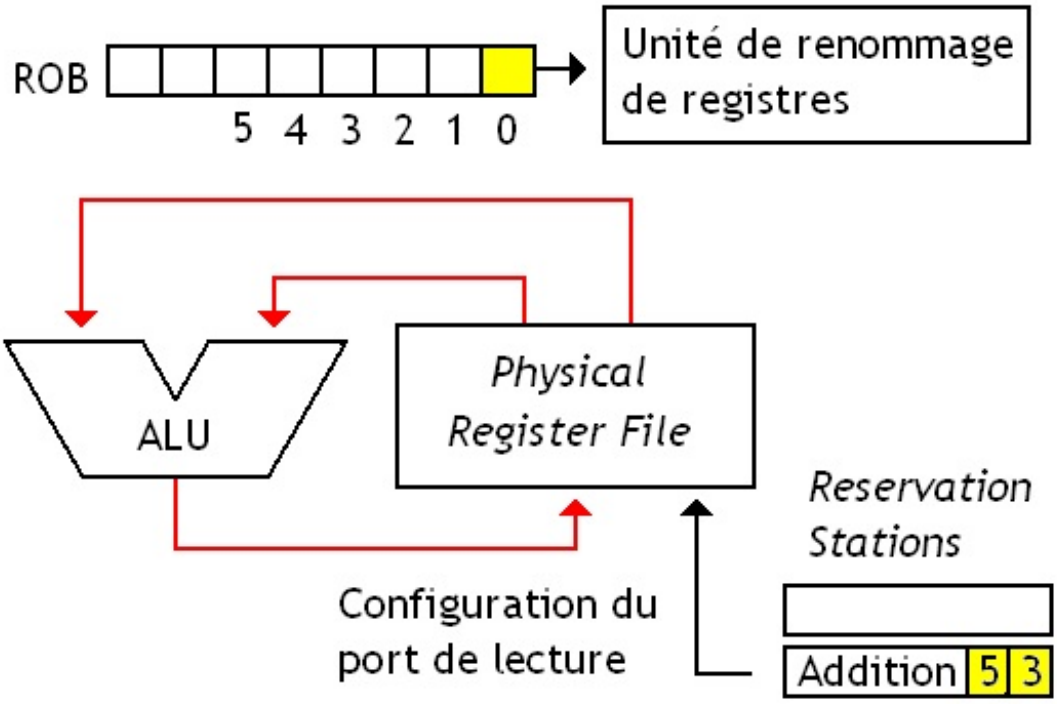


Quand le *Reorder Buffer* veut écrire une donnée en mémoire, il a juste à la lire depuis le *Rename Register File*, et l'écrire directement dans le *Retiremment Register File*. Pareil pour les *Reservation Stations* : quand toutes les opérandes d'une instruction sont disponibles, elles sont lues depuis le *Rename Register File* ou le *Retiremment Register File*, et envoyées sur l'entrée de l'unité de calcul.

Le pipeline du processeur reste identique avec la version du renommage utilisant un *Reorder Buffer*, à part que l'étape de lecture des opérandes s'appelle maintenant *Register Read*.

Physical Register File

On peut encore améliorer la structure vue au-dessus en utilisant non pas deux *Register Files*, mais un seul. On utilisera alors un grand *Register File* qui contient à la fois les registres physiques que les registres virtuels. On appelle celui-ci un ***Physical Register File***. Encore une fois, les *Reservations Stations* et le *Reorder Buffer* ne stockent plus les opérandes de l'instruction, mais seulement le nom du registre associé à la donnée.



Aujourd 'hui, presque tous les processeurs utilisent ce genre de structure pour faire du renommage de registres. Il faut dire qu'un *Physical Register File* consomme bien moins d'énergie que ses concurents. Il faut dire que les données ne sont pas copiées d'un *Rename Register File* vers un *Retiremment Register File*. Et de plus, les diverses structures matérielle associées, comme le *Reorder Buffer*, contiennent des pointeurs vers nos registres, ce qui prend moins de place que stocker directement les données, plus longues. C'est pour cela que cette amélioration est utilisé sur les derniers processeurs actuels.

Autre avantage : certaines opérations deviennent inutiles si le renommage de registres est fait intelligemment. Il s'agit des instructions de copie d'un registre dans un autre (les `mov`) ou d'échange entre deux registres (`xchg`). Elles peuvent être supprimées via une application correcte du *register renamming*. Après la copie de notre registre dans un autre, le contenu des deux registres est identique. Ce n'est que lors d'une écriture à lieu dans des deux registres que leurs contenus différents. Au lieu de faire cette recopie, on peut utiliser un seul registre physique pour stocker la valeur à recopier et renvoyer toute lecture des deux registres architecturaux vers ce registre physique. Lors d'une écriture dans un de ces deux registres, il suffira d'utiliser un registre physique vide pour stocker la donnée à écrire et l'associer avec le nom de ce registre. On peut considérer qu'il s'agit d'une forme de *Copy On Write*, appliquée sur des registres, en considérant que le nom d'un registre est un pointeur (il vaudrait mieux parler de référence, mais bon...) vers un registre physique.

L'unité de renommage

Dans nos processeurs, les registres sont identifiés par ce qu'on appelle un **nom de registre**. Ce terme doit vous rappeler quelque chose, et ce n'est pas un hasard : on a vu cela dans le chapitre sur l'assembleur. Quoiqu'il en soit, nos registres sont donc identifiés par ces noms de registres qui ne sont autre que des suites de bits dont la valeur va permettre d'identifier un registre architectural parmi tous les autres. Nos registres physiques sont eux aussi identifiés par un nom de registre, mais qui est seulement connu du processeur : le ***Tag***. Pour attribuer un registre architectural à un registre physique, **il suffit de remplacer le nom du registre architectural par le *tag* du registre physique qui lui est attribué**. On dit alors que le registre architectural est renommé.

Dans un processeur utilisant le renommage de registres, il y a bien plus de registres physiques que de registres architecturaux. Donc, le *tag* d'un registre physique est plus grand que le nom d'un registre architectural.

Register Map Table

Ce remplacement est effectué par un circuit spécialisé, casé dans un étage supplémentaire du pipeline. Celui-ci va prendre sur ses entrées le nom des registres architecturaux à manipuler, et va fournir sur sa sortie les *tag* des registres physiques correspondant. Ce circuit s'appelle la ***Register Map Table***. Ce circuit est conceptuellement divisé en deux gros sous-circuits. Je dis conceptuellement, parce que ces deux circuits sont parfois fusionnés en un seul, mais dans les explications qui vont suivre, on fera comme si on disposait de deux circuits bien séparés.

Free List

Pour commencer, nous allons prendre le premier de ces sous-circuits la ***Free List***. Celle-ci sert à renommer le ou les registres modifiés par notre instruction. En bref, ceux dans lesquels notre instruction va aller écrire. Renommer ces registres est très simple. Pour éviter que le registre de destination soit un registre déjà utilisé par une autre instruction (voire par l'instruction elle-même, si le registre de destination est un registre source), on doit prendre un registre physique inutilisé à la place. Notre *Register Map Table* doit donc garder une liste des registres vides , utilisables librement par nos instructions. Pour cela, on garde cette liste des registres inutilisés dans une petite mémoire qu'on appelle la *Free List*.

Lorsque la donnée contenue dans un registre n'a plus à être utilisées par les prochaines instruction, celle-ci devient inutile. On peut alors réutiliser ce registre comme bon nous semble. A ce moment, ce registre passe dans la *Free List*. Détecter les registres réutilisables est assez complexe, et peut en plus se faire de plusieurs façons, suivant le nombre d'instruction démarrées simultanément. Si ce nombre est faible, on peut détecter facilement les registres physiques devenus réutilisables. Il suffit que l'instruction qui a écrit le résultat dedans quitte le *Re-Order Buffer*. A ce moment, ce résultat est copié du Registre virtuel vers le *Retiremment Register File*. Toute lecture du résultat sera fournie par le *Retiremment Register File*, et pas par le registre physique. On peut donc le réutiliser comme bon nous semble. Par contre, si on peut *Issue* un grand nombre d'instructions par cycle d'horloge, la situation est plus complexe.

Register Alias Table

Maintenant que l'on a réussit à renommer les registres de destination de notre instruction, on doit renommer les registres sources, qui contiennent les opérandes. C'est le rôle qui sera dévolu à la ***Register Alias Table***. Il s'agit d'une mémoire qui contient, pour chaque registre architectural, le registre physique associé. Elle se charge donc de faire la correspondance entre nom d'un registre source, et *Tag* d'un registre physique.

Cette correspondance est mise à jour à chaque fois qu'un registre vide est réquisitionné pour servir de registre de destination pour une instruction. A ce moment, la *Register Alias Table* est mise à jour : on se souviendra alors à quel registre physique correspondra le registre architectural dans lequel on stocke le résultat de notre instruction. Et il suffira de réutiliser ce nom de registre physique par la suite.

Cependant, le résultat ne sera pas immédiatement disponible pour notre instruction. Il ne le sera qu'après avoir été calculé. C'est seulement à ce moment là que notre registre physique contiendra une donnée valide et sera utilisable par d'autres instructions. Pour savoir si notre registre contient une donnée valide, notre *Register Alias Table* va contenir pour chaque registre architectural, un bit de validité. Ce dernier indiquera si le registre virtuel correspondant contient la donnée voulue. Ce bit de validité est mis à jour lors de l'écriture du résultat dans le registre physique correspondant, et cette mise à jour serra propagée dans le reste du processeur. Cela permet ainsi de savoir quand les données d'une instruction sont prêtes.

Détails

Sur les processeurs utilisant un *Physical Register File*, on n'a pas à aller plus loin. Avec ce qu'on a dit plus haut, pas besoin de rajouter quoi que ce soit pour effectuer du renommage de registres. Mais pour les autres processeurs, c'est autre chose. Il faut dire que sur les autres processeurs, on dispose de deux *Register File* : un pour les registre architecturaux, et un pour les registres physiques. Et on doit préciser si on doit lire une donnée depuis les registres architecturaux ou depuis les registres physiques. Quand on doit lire une donnée depuis les registres physiques, on n'a strictement aucune correspondance entre le registre architectural et un registre physique. La *Register Alias Table* a donc une entrée de vide, ce qui est indiqué d'une facon ou d'une autre à l'instruction à renommer.

Petite remarque : en cas de mauvaise prédiction de branchement ou d'exception matérielle, on doit remettre notre *Register Alias Table* et la *Free List* dans l'état dans lequel elle était avant le chargement de l'instruction fautive. Cela peut se faire de différentes manières, mais la plus courante est de stocker dans le *Re-Order Buffer* ce qui a été modifié dans l'unité de renommage de registres par l'instruction correspondante. Ainsi, lorsqu'une instruction sera prête à *commit*, et qu'une exception ou mauvaise prédiction de branchement aura eu lieu, les modifications effectuées dans l'unité de renommage de registres seront annulées les unes après les autres. Une autre solution consiste à garder une copie valide de la *Register Alias Table* dans une mémoire à part, pour al restaurer au besoin. Par exemple, si jamais notre *Register Alias Table* détecte un branchement, son contenu est sauvegardé dans une mémoire intégrée au processeur.

Implémentation

Il existe deux façons pour implémenter une *Register Alias Table*. La plus ancienne consiste à utiliser une mémoire associative. Cette mémoire peut être vu comme une sorte de mémoire cache un peu spéciale. Cette mémoire associative stocke des correspondances entre registres architecturaux, et registres physiques. Son *Tag* correspond à un nom de registre architectural, tandis que la ligne de cache qui correspond contient le nom de registre virtuel associé. Quand on veut obtenir le registre physique qui correspond à un registre architectural, il suffit de comparer ce nom du registre architectural avec les *Tags* du cache et de retourner le nom de registre physique si on a correspondance avec un *Tag*. Mais il faut faire attention au cas où plusieurs instructions écrivent dans le même registre architectural à des moments différents. Et c'est là que l'on s'aperçoit de la différence avec un cache : dans ce cas, cette mémoire associative va conserver plusieurs correspondances registre architectural - registre physique. Dans ce cas, on doit renvoyer l'entrée de l'instruction la plus récente parmi toutes celles qui correspondent. Pour ce faire, nos correspondances sont triées par ordre d'arrivée.

Sur les processeurs plus récents, on implémente notre *Register Alias Table* autrement. On utilise une mémoire RAM, dont les adresses correspondent aux noms de registres architecturaux, et dont le contenu d'une adresse correspond au nom du registre physique associé. C'est plus rapide et utilise moins de transistors. Mais dans ce cas, on n'a qu'une seule correspondance entre registre physique et registre architectural. Cela ne pose pas vraiment de problèmes, si on renomme nos instructions dans l'ordre.

Les optimisations des accès mémoire

Dans tout ce qu'on a vu précédemment, on a surtout parlé des instructions arithmétiques et des branchements. Le chapitre sur l'exécution *Out Of Order* nous a montré que l'on pouvait modifier l'ordre des instructions pour gagner en efficacité. Et avec le chapitre sur le renommage de registre, on a vu comment supprimer certaines dépendances entre instructions lorsque ces instructions utilisant des registres. Ainsi, si deux instructions réutilisaient le même registre, mais à des instants différents, on pouvait supprimer les dépendances WAR et WAW qui en résultait.

Le seul problème, c'est que tout cela n'est valable que pour les instructions travaillant sur des registres. Si nos instructions doivent aller lire ou écrire dans la mémoire, le renommage de registre ne servira à rien ! Dans ce cas précis, on ne peut donc supprimer les dépendances WAR et WAW avec ce genre de techniques, ce qui diminue les possibilités d'exécution *Out Of Order*. Pour améliorer la situation, il a fallu trouver un moyen de limiter les effets de ces dépendances de données entre instructions d'accès mémoires. Pour ce faire, les concepteurs de processeurs et les chercheurs en architecture des ordinateurs ont inventés diverses techniques plus ou moins efficaces permettant de gérer ces dépendances entre instructions mémoires. Ces techniques sont ce qu'on appelle des techniques de ***Memory Disambiguation***.

Dépendances, le retour !

Modifier l'ordre d'exécution des accès à la mémoire est une chose assez efficace en terme de performances. Par exemple, il vaut mieux effectuer les lectures le plus tôt possible. Il faut dire que ces lectures prennent un certain temps : accéder à une donnée ne se fait pas immédiatement. Cela peut prendre moins de 10 cycles d'horloge pour un accès au cache L1, mais peut facilement monter et atteindre des nombres à 2 chiffres pour les accès au L2, et trois chiffres pour les accès à la mémoire. Autant dire que ne rien faire durant cet accès au cache ou à la mémoire, et faire attendre les instructions suivant une instruction de lecture n'est pas une bonne chose. L'idéal serait d'exécuter des instructions indépendantes de l'accès en mémoire pendant qu'on attend que la donnée voulue soit lue. Pour ce faire, il suffit d'exécuter la lecture le plus précocement possible, et éxecuter des instructions indépendantes pendant ce temps.

Seul problème : il faut que toutes les instructions ayant une dépendance avec cette lecture aient déjà finies de s'exécuter avant qu'on puisse lancer la lecture. Si on se trouve dans un tel cas, il se peut que l'on ne puisse démarrer notre lecture aussi tôt que prévu, parce qu'une instruction ayant une dépendance avec notre lecture n'est pas terminée : impossible de faire passer notre lecture avant celle-ci. Reste à savoir si ces dépendances sont monnaie courante.

Utilité

Dans le chapitre précédent, on a vu que ces dépendances naissaient lorsque l'on des instructions différentes veulent lire ou écrire dans des emplacements mémoire identiques. Dans les cas des registres, cela arrive très souvent : un processeur possède souvent une faible quantité de registres, qui doit donc être utilisée au mieux. Ainsi, nos compilateurs n'hésitent pas à réutiliser des registres dès que possible, et n'hésitent pas à écraser des données qui ne sont plus nécessaires pour stocker des résultats utiles. Réutiliser des registres le plus possible fait donc apparaitre de nombreuses dépendances WAR et WAW. Mais pour les accès mémoires, c'est autre chose. Accéder à la mémoire n'est pas rare, certes, mais réutiliser de la mémoire l'est. Il est en effet très rare qu'on doive lire ou écrire à des adresses identiques dans un cours laps de temps, et rares sont les dépendances WAR et WAW. Il faut dire que ces situations correspondent souvent à des données qui sont stockées temporairement sur la pile, à cause d'un manque de registres. On pourrait donc croire que chercher à supprimer les dépendances WAR et WAW pour les accès à la mémoire ne servirait que marginalement, et ne serait donc qu'un coup d'épée dans l'eau.

Mais la situation est beaucoup plus compliquée que ce que cette présentation naïve vous l'a laissé entendre.

De nouvelles dépendances

Le seul truc, c'est que notre processeur ne peut pas toujours savoir si deux accès à la mémoire vont se faire au même endroit ou pas. Pour les instructions utilisant l'adressage absolu (l'adresse à laquelle lire ou écrire est stockée dans la suite de bits représentant notre instruction), il n'y a pas de problèmes. Mais le seul truc, c'est que ce n'est pas le cas pour d'autres modes d'adressages. Par exemple, il n'est pas rare que nos adresses soient stockées dans des registres. Il est en effet monnaie courante de ne pas connaitre à l'avance les adresses à laquelle lire ou écrire, et calculer des adresses est une chose commune de nos jours. Dans des cas pareils, il est impossible de savoir si deux accès à la mémoire se font à la même adresse ou pas.

Bilan : deux accès à la mémoire peuvent être totalement indépendants, mais le processeur ne peut pas le savoir. Résultat : il est obligé de supposer par sécurité que ces deux accès sont dépendants, ce qui va limiter ses possibilités. Il ne pourra pas changer l'ordre de ses instructions pour gagner en efficacité. Et cela arrive très souvent : presque à chaque accès mémoire !

Il faut noter que ce genre de situations arrive aussi dans un domaine assez éloigné. Certains compilateurs doivent faire face à un problème similaire dans certaines situations : dans certaines conditions, ils ne savent pas si deux adresses mémoire utilisées dans un programme sont différentes ou pas. Et dans ces conditions, ils doivent éviter de modifier l'ordre des accès à ces adresses, ce qui limite grandement les possibilités d'optimisation. C'est ce qu'on appelle le phénomènes d'***aliasing* des pointeurs**.

Autre problème : il arrive qu'il ne soit pas possible de déplacer une lecture avant une autre. Cela arrive dans un cas simple : quand on charge une adresse dans un registre et qu'on chercher à lire le contenu de cet adresse. On a obligatoirement besoin de charger l'adresse avant de charger la donnée pointée par cette adresse. La première lecture doit se faire avant l'autre. Une autre dépendance fait son apparition.

Dépendances de nommage

Pour limiter la catastrophe, notre processeur va utiliser des mécanismes permettant de diminuer les effets de nos dépendances, en supprimant ou atténuant celles-ci. Divers mécanismes de *Memory Disambiguation* ont ainsi étés inventés.

Ces techniques sont basées sur un principe assez proche de celui qui est derrière le renommage de registres. Écrire notre données au même endroit que les lectures ou écritures précédentes va faire apparaitre des dépendances. Pour supprimer ces dépendances, on va simplement écrire la donnée ailleurs, et attendre que les lectures ou écritures précédentes à cette adresse mémoire soient terminées, avant de déplacer notre donnée au bon endroit. Pour ce faire, on pourrait penser à utiliser du renommage d'adresses mémoires. Mais ce serait compliqué, et augmenterait le nombre d'écritures en mémoire d'une façon assez importante. Même si certains processeurs utilisent cette technique, elle est tout de même assez peu utilisée, du fait de ses défauts. D'autres solutions existent, plus efficaces.

Store queue

Si on utilisait le renommage d'adresses mémoires, on devrait utiliser la mémoire pour stocker temporairement des données à écrire, ces donnée venant d'instructions exécutées en avance comparé à ce qui était prévu dans l'ordre du programme. Au lieu d'utiliser la mémoire, on préfère écrire ces données dans une mémoire intégrée au processeur, spécialement conçue dans ce but : la ***store queue***. Et oui, j'ose vous sortir un terme barbare sans prévenir : il faudra vous y habituer.

Ainsi, on peut enregistrer nos données dans cette Store Queue temporairement. Chaque donnée présente dans cette Store Queue va quitter celle-ci pour être enregistrée en mémoire sous une condition bien particulière. Si dans l'ordre des instructions du programme, on trouve des lectures ou des écritures qui lisent ou modifient l'adresse à laquelle on veut écrire cette donnée, alors on doit attendre que celles-ci soient terminées pour pouvoir démarrer l'écriture de notre donnée et la faire quitter la *Store Queue*.

L'utilisation d'une *Store Queue* a d'autres avantages : on peut éxecuter notre écriture avant certaines instructions qui pourraient lever une exception matérielle. Par exemple, on peut effectuer notre écriture avant une opération de division, qui peut potentiellement lever une exception de division par zéro. Sans *Store Queue*, on aurait exécutée notre écriture et modifié la mémoire alors que notre écriture n'aurait jamais été exécutée, vu que l'exception levée par la division aurait alors interrompu le programme et aurait parfaitement pu faire zapper notre écriture sous certaines circonstances. Avec une *Store Queue*, il suffit d'effacer la donnée à écrire de la *Store Queue*, ainsi que les informations qui lui sont associée, sans rien écrire dans la mémoire.

Cette *Store Queue* est localisée dans les unités *Store*, qui se chargent d'effectuer les opérations d'écriture en mémoire.

Bypass Store Queue

Mais utiliser bêtement une *Store Queue* sans réfléchir risque de poser quelques problèmes. Imaginez la situation suivante : on écrit une donnée dans la *Store Queue*, et on cherche à lire cette donnée quelques cycles plus tard. La donnée n'a alors pas encore été écrite dans la mémoire RAM, et est encore dans la *Store Queue*. La lecture renverra alors la donnée qui est en mémoire, soit une donnée différente de celle qui est dans la *Store Queue*. En clair, le programme se met à faire n'importe quoi...

Pour éviter ce genre de petits désagréments, on doit permettre de lire des données dans la *Store Queue*. Ainsi, si on veut effecteur une lecture, les unités en charge des lectures vont ainsi aller chercher la donnée au bon endroit : dans la *Store Queue*, ou dans la mémoire. Cette solution, cette technique s'appelle le ***Store To Load Forwarding***.

Pour implémenter cette technique, on construit souvent notre *Store Queue* sous la forme d'une sorte de mémoire cache, contenant les données à écrire dans la mémoire. Cette mémoire cache a pour tag l'adresse à laquelle on cherche à écrire notre donnée. Si jamais une lecture a lieu, on va d'abord vérifier si une écriture à la même adresse est en attente dans la *Store Queue*. Si c'est le cas, alors on renvoie la donnée présente dans celle-ci. Sinon, on va lire la donnée en mémoire RAM.

Au fait : que se passe-il si jamais deux données sont en attente d'écriture dans la *Store Queue* et qu'on souhaite effectuer une lecture à la même adresse ?

Et bien dans ce cas là, la *Store Queue* se charge de donner la dernière donnée à avoir été écrite, histoire de renvoyer la donnée la plus à jour. En exécutant les lectures dans le bon ordre, cela ne pose aucun problème.

Dependances d'alias

Comme je l'ai dit plus haut, le processeur ne peut pas toujours savoir si deux accès mémoires vont se faire au même endroit ou non. A cause de cela, des dépendances RAW, WAR, et WAW fictives apparaissent inutilement. Alors certes, l'utilisation d'une

Store Queue permet de supprimer ces dépendances WAR et WAW. Mais pour supprimer les fausses dépendances RAW, il faut trouver d'autres solutions.

Vérifications des adresses

La première solution pour limiter les catastrophes consiste à séparer les accès à la mémoire en deux parties, en deux micro-instructions :

- une qui calcule l'adresse à laquelle on doit lire ou écrire : on gère donc le mode d'adressage durant cette étape ;
- et une autre qui va accéder à la mémoire une fois l'adresse calculée.

Ainsi, si on veut exécuter une lecture de façon anticipée, il suffit de calculer son adresse, et vérifier si toutes les écritures précédant cette lecture dans l'ordre du programme se font à une adresse différente. Si c'est le cas, c'est que notre lecture va lire une donnée qui ne sera pas modifiée par les écritures en cours ou en attente dans la *Store Queue*. On peut donc l'effectuer directement. Dans le cas contraire, on doit attendre la fin des écritures à la même adresse pour pouvoir démarrer la lecture en mémoire ou dans la *Store Queue*.

Cette technique peut s'implémenter facilement avec la *Store Queue* qu'on a vue au-dessus. Cependant, cette façon de faire n'est pas des plus efficace : certaines opportunités d'optimisation sont perdues. En effet, une écriture ne peut pas être placée dans la *Store Queue* tant que l'adresse de la donnée à écrire n'est pas connue. Elle reste alors bloquée dans l'*Instruction Window* ou dans les unités de décodage, en bloquant toutes les autres instructions plus récentes, même si celles-ci n'ont pas la moindre dépendance. Autant dire que cela fait quelques instructions bloquées qui auraient pu s'exécuter. Heureusement, ce n'est pas la seule méthode : on peut aussi utiliser des **matrices de dépendances**. Ces matrices de dépendances permettent de repérer de façon optimale toutes les dépendances entre accès mémoires, sans en laisser passer une seule.

Matrices de dépendances

Ces matrices forment une espèce de tableau carré, organisé en lignes et en colonnes. Chaque ligne et chaque colonne se voit attribuer une instruction. A l'intersection d'une ligne et d'une colonne, on trouve un bit. Celui-ci permet de dire si l'instruction de la ligne et celle de la colonne ont une dépendance. Si ce bit est à 1, alors l'instruction de la ligne a une dépendance avec celle de la colonne. Si ce bit est à zéro, les deux instructions sont indépendantes. Cette technique nous permet donc de comparer une instruction avec toutes les autres, histoire de na pas rater la moindre dépendance. A chaque ligne, on attribue une lecture ou une écriture.

Commençons par voir la version la plus simple de ces matrices de dépendances. Avec celles-ci, on vérifie juste si toutes les adresses des écritures précédentes sont connues ou non. Si elles ne sont pas toutes connues, les lectures vont attendre avant de pouvoir s'exécuter. Dans le cas contraire, on peut alors démarrer nos accès mémoires. La vérification des dépendances (est-ce que deux accès mémoires se font à la même adresse) se fait alors avec une *Store Queue* ou dans des circuits spécialisés.

Lorsque le processeur démarre une écriture dont il ne connait pas l'adresse de la donnée à écrire, il va d'abord insérer cette écriture dans ce tableau carré dans une ligne. Cette ligne sera celle d'indice i. Puis, il va mettre tous les bits de la colonne de même indice (i) à 1. Cela permet de dire que notre écriture peut potentiellement avoir une dépendance avec chaque instruction en attente. Vu qu'on ne connait pas son adresse, on ne peut pas savoir. Lorsque cette adresse est alors connue, les bits de la colonne attribuée à l'écriture sont remis à zéro. Quand tous les bits d'une ligne sont à zéro, la lecture ou écriture correspondante est envoyée vers les circuits chargés de gérer les lectures ou écritures. Ceux-ci se chargeront alors de vérifier les adresses des lectures et écritures, grâce à une *Store Queue* et le *Store-to-Load Forwarding* associé.

Cette technique peut être améliorée, et gérer la détection des dépendances elle-même, au lieu de les déléguer à une *Store Queue*. Dans ce cas, on doit commencer par ajouter l' adresse à laquelle notre instruction va lire ou écrire pour chaque ligne. Puis, à chaque fois qu'une adresse est ajoutée dans une ligne, il suffit de la comparer avec les adresses des autres lignes et mettre à jour les bits de notre matrice en conséquence.

Efficacité

Cette technique n'est pas très efficace : il est en effet peu probable que toutes les adresses des écritures précédant une lecture soit connue lorsque l'on veut lire notre donnée. Autant dire que cette technique n'est pas utilisée seule, et elle est complétée par d'autres techniques plus ou moins complémentaires.

Exécution spéculative

Les techniques vues précédemment donnent des résultats assez bons, et sont des techniques assez utiles. Mais c'est loin d'être la seule technique qui aie été inventée pour diminuer l'effet des dépendances RAW. Parmi ces autres techniques, nombreuses sont celles qui utilisent l'**exécution spéculative**. Le terme peut paraitre barbare, mais il consiste simplement à exécuter des

instructions de façon anticipée, en supposant certaines choses, et en remettant le processeur à zéro si la supposition se révèle être fausse. La prédiction de branchement est un cas d'exécution spéculative assez connu.

Dans le cas des lectures et écritures en mémoire, rien ne nous empêche de réorganiser spéculativement l'ordre des lectures et des écritures quelque soit la situation, même si on ne sait pas si les dépendances RAW entre deux accès mémoires sont fictives ou réelles. Alors bien sûr, on ne doit pas faire cela bêtement : notre processeur va vérifier si jamais il a fait une erreur en exécutant une lecture trop anticipée, en vérifiant les adresses auxquelles il va écrire et lire. Si jamais le processeur a exécuté une lecture trop tôt (avant une écriture à la même adresse dans l'ordre du programme), il va se charger d'éliminer les modifications qui ont étés faites par cette lecture foireuse, et reprendra l'exécution du programme convenablement.

Pour pouvoir fonctionner correctement, notre processeur doit donc vérifier qu'il n'y a pas d'erreur. Une erreur correspond à une lecture à la même adresse qu'une écriture, avec la lecture qui est placée après l'écriture dans l'ordre des instructions imposé par le programme. Pour faire ces vérifications, le processeur va garder la trace des lectures effectuées dans une sorte de mémoire cache, la ***Load Queue***. Cette *Load Queue* va conserver pour chaque lecture : l'adresse de la lecture effectuée, ainsi que la donnée lue. Cette *Load Queue* va conserver ces informations durant un certain temps, jusqu'à ce que toutes les instructions précédant la lecture dans l'ordre du programme soient terminées. Une fois que ces instructions sont finies, la lecture est effacée de la *Load Queue*, ou tout simplement oubliée.

Ainsi, à chaque écriture, il suffit de vérifier si une lecture a accédé à la même adresse se trouve dans cette *Load Queue*. Rien de bien compliqué : on compare l'adresse de l'écriture avec les tags de la *Load Queue*, qui contiennent l'adresse à laquelle notre lecture a accédé. Si on ne trouve pas de correspondance, alors il n'y a pas d'erreurs. Mais si c'est le cas, on doit alors supprimer du pipeline toutes les instructions exécutées depuis la lecture anticipée fautive. En clair, on fait exactement la même chose que lors d'une mauvaise prédiction de branchement : on vide le pipeline. Petit détail : sur certaines processeurs, la *Load Queue*, et la *Store Queue* sont fusionnées dans une seul gros circuit.

Memory Dependance Prediction

Certains processeurs utilisent des mécanismes encore plus poussés pour éviter le plus possible de passer outre des dépendances RAW. Pour cela, ils essayent de prédire si deux accès à la mémoire se font à la même adresse, et si ils ont une dépendance RAW. Pour implémenter cette technique, certains processeurs incorporent une unité qui va fonctionner comme une unité de prédiction de branchement, à la différence qu'elle va chercher à prédire les dépendances entre instructions. Si cette unité prédit que deux accès mémoires sont indépendantes, le processeur se permet de changer leur ordre. Mais si ce n'est pas le cas, il exécute nos deux instructions dans l'ordre imposé par le programme et les dépendances. Ainsi, on pourra alors exécuter spéculativement une lecture si cette unité nous dit : "j'ai prédit qu'il n'y aura pas d'écritures à l'adresse que tu veux lire".

Bon, évidemment, il faut bien prendre en compte le cas où cette unité de prédiction des adresses mémoires se plante, et on gère cela comme pour les mauvaises prédictions de branchement : on vide le pipeline pour éviter que cette lecture anticipée ne vienne mettre le bazar dans notre pipeline.

Bien sur, cela ne permet pas de se passer des techniques vues auparavant, mais cela aide tout de même à éviter de payer les couts des erreurs sur des dépendances RAW.

Wait Table

Une des techniques les plus simple pour prédire les dépendances d'alias est celle de la ***Wait Table***. Le principe de cette technique est diablement simple : si jamais une lecture/écriture va aller trifouiller la même adresse qu'une de ses consœurs, on s'en souviens pour la prochaine fois.

Pour s'en souvenir, il suffit de rajouter quelque chose qui permettent de se souvenir des instructions mémoire ayant une dépendance RAW avec une de leur consœur. Pour cela, il suffit de rajouter une mémoire dans laquelle on va placer les adresses (le *Program Counter*) des instructions ayant une dépendance. L'adresse de chaque instruction qui s'exécute va ainsi être stockée dans cette mémoire, et va être associée à un bit. Ce bit vaudra 0 si l'instruction correspondante n'a aucune dépendance, et 1 dans le cas contraire. L'unité de décodage pourra alors déduire les dépendances de nos instructions. C'est cette technique qui était utilisée sur les processeurs Alpha 21264. Pour éviter les faux positifs, le bit associé à une instruction était remis à zéro au bout d'un certain temps. Tous les 100 000 cycles d'horloge, pour être précis.

On peut aussi améliorer l'idée en se passant du bit, et en ne stockant dans cette mémoire cache que les instructions ayant une dépendance, et pas les autres. Et là encore, on peut décider de virer une instruction de cette mémoire au bout d'un certain temps pour éviter les faux positifs.

Autres

La technique vue au-dessus a un sacré défaut. Si jamais une instruction mémoire est indiquée comme ayant une dépendance avec une autre, on ne sait pas avec quelle instruction elle a cette dépendance. Notre instruction doit donc attendre que toutes les lectures et écritures qui la précédent soient terminée avant de pouvoir s'exécuter. Même celles avec laquelle notre instruction

n'a aucune dépendance. Nos instructions peuvent donc se retrouver à attendre inutilement avant de pouvoir s'exécuter. Pour éviter cela, d'autres techniques de prédiction de dépendances mémoires ont étés inventées.

Load Adress Prediction

Comme on peut s'en douter, connaitre l'adresse d'une lecture à l'avance permet de fortement limiter la casse, et permet à nos mécanismes de *Memory Disambiguation* de fonctionner plus efficacement. Si jamais cette adresse est connue plus tôt, on détecte les dépendances plus rapidement et on peut agir en conséquence. Reste que le calcul de cette adresse ne se fait pas tout seul. Il faut que les opérandes de ce calcul soient disponibles. Et ce n'est pas garantit. Il arrive que certaines instructions doivent attendre que l'adresse à lire/écrire soit calculée. Et pendant ce temps, le processeur peut faire des ravages en spéculant trop.

Pour rendre ce calcul d'adresse plus rapide, on peut améliorer les circuits chargés du calcul de ces adresses, mais on peut aussi faire pire. On peut spéculer sur l'adresse d'une lecture ou écriture ! Tenter de prédire à l'avance cette adresse peut améliorer les performances assez facilement. Reste à savoir comment faire pour prédire cette adresse. Et encore une fois, on peut tenter de prédire celle-ci en utilisant des régularités de nos accès mémoires.

Last Adress

Commençons par aborder la première technique. Celle-ci est très simple : il suffit de supposer que chaque instruction de lecture accède toujours à la même adresse.

Pourquoi ça marche ?

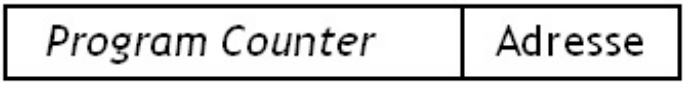
On pourrait se demander pourquoi une lecture ou écriture irait accéder plusieurs fois à la même adresse. Pour répondre à cela, il faut savoir que nos programmes sont parfois obligés d'accéder à la même adresse à cause du compilateur.

Il arrive que les compilateurs n'arrivent pas à gérer efficacement les accès mémoires. Diverses raisons existent pour cela : certaines dépendances entre instructions forcent certaines données à être relues depuis la mémoire. Cela arrive notamment lorsque l'on utilise des pointeurs ou des références : divers phénomènes complexes d'*aliasing* des pointeurs peuvent générer des relectures intempestives de données en mémoire. Cela peut aussi venir de machins qui arrivent lorsqu'on compile du code qui provient de plusieurs librairies, bref.

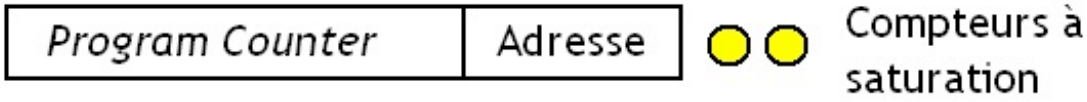
Implémentation

Pour implémenter cette technique, rien de plus simple : il suffit de stocker un historique pour chaque instruction dans une petite mémoire. Cette mémoire cache stockera l'adresse accédée pour chaque instruction mémoire récemment utilisée. Pour faire l'association instruction <-> adresse lue/écrite, il suffit de mettre l'adresse de notre instruction (le *Program Counter*) dans le *Tag* associée à une adresse.

Voici donc à quoi ressemble une ligne de cache de cette petite mémoire.



Bien sûr, il est rare qu'une instruction de lecture ou d'écriture accède à la même case mémoire plusieurs fois de suite. On doit donc trouver un moyen de savoir si notre instruction accède plusieurs fois à la même adresse ou pas. Pour cela, on ajoute des compteurs à saturation pour chaque instruction (chaque ligne de cache). Ces compteurs sont incrémentés à chaque fois qu'une instruction réutilise la même adresse, et décrémenté en cas de changements. Vu que les instructions qui accèdent toujours à la même adresse sont rares, il est préférable d'initialiser ces compteurs de façon à ce qu'ils disent que toute nouvelle instruction change d'adresse.



Stride

Autre méthode pour prédire l'adresse d'une lecture/écriture. Supposer que cette lecture/écriture va accéder à des adresses séparées par des intervalles réguliers.

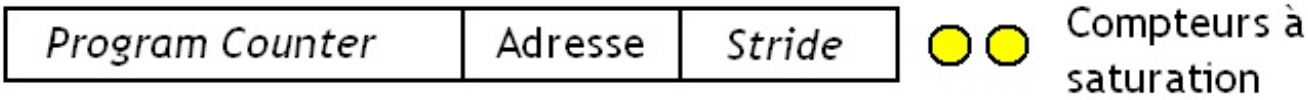
Les accès à des adresses consécutives rentrent dans ce cadre. Mais les accès mémoires sur des adresses séparées par une distance fixe sont aussi pris en compte. Cette distance entre deux adresses, on l'appelle le ***Stride***.

Pourquoi ça marche ?

Ce genre d'accès doit vous rappeler quelque chose. Si vous êtes arrivé jusqu'à ce chapitre, vous devez surement vous souvenir du chapitre sur le *Prefetching*. Et on y a vu que des accès mémoire de ce type provenait de l'utilisation des tableaux. Quand on parcourt ceux-ci, on accède à la mémoire de cette façon. Et cette fois, on sort vraiment d'artillerie lourde. Il faut dire que ce genre d'accès à des tableaux est vraiment courant et que beaucoup de programmes sont concernés. Prédire les adresses de cette façon peut donc donner lieu à des gains pas vraiment négligeables.

Implémentation

Reste à savoir comment implémenter cette technique dans notre processeur. Et il n'y a rien de plus simple : il suffit de reprendre notre mémoire vue au-dessus, et d'y rajouter de quoi stocker le *Stride* pour chaque adresse. Ce *Stride* sera la distance entre une adresse, et celle accédée précédemment par notre instruction.



Ce *Stride* est déterminé par notre circuit chargé de la prédiction. Celui-ci garde en mémoire la dernière adresse accédée par notre instruction, et il fait la différence avec l'adresse lue. Il en déduit le *Stride*, et stocke celui-ci dans notre mémoire cache. A chaque accès, ce *Stride* est ajouté à l'adresse contenue dans notre cache. L'ancienne adresse dans le cache est remplacée par la nouvelle une fois qu'on dispose de l'adresse valide.

Context Based Predictor

Certains prédicteurs d'adresse se permettent de faire un tout petit peu mieux. Ceux-ci sont capables de repérer des accès mémoires qui se répètent de façon régulière et cyclique, même s'ils n'ont aucun *Stride*.

Ce genre d'accès se trouve assez souvent lorsque l'on manipule des listes chainées ou des structures de données assez irrégulières comme des arbres, des graphes, etc.

Pour gérer ces accès, on stocke les dernières adresses accédées dans une petite mémoire cache. Cette mémoire cache stockera une instruction par ligne, dont l'adresse sera placée dans le *Tag*. La ligne de cache associée contiendra les dernières adresses accédées par l'instruction. Voici à quoi ressemble une ligne de cache de cette mémoire.

Le tout est complété par une unité qui se charge de déterminer laquelle de ces adresses est la bonne. Le tout est ensuite complété par une unité chargée de mettre à jour la mémoire cache qui contient les adresses de chaque instruction. L'implémentation de ces unités peut fortement varier suivant les processeurs, aussi je ne rentrerais pas dans les détails.

Efficacité

A ce stade, je dois préciser que cette technique n'est pas encore tout à fait mature, et qu'aucun processeur ne l'implémente encore. En tout cas, la recherche sur le sujet est encore en cours, et même si aucun processeur n'a encore implémenté de technique de ce genre, cela ne saurait tarder. Quoiqu'il en soit, ces unités de prédiction sont tout de même utilisées dans d'autres circonstances. Des variantes de ces unités de prédiction d'adresse sont utilisées dans les *Prefetchers*, ceux utilisées pour précharger des données depuis le cache de donnée. Et leur efficacité est assez bonne, voire excellente dans certains cas. Mais pour le moment, ces unités de prédiction d'adresse ne sont pas encore utilisées pour prédire les adresses à lire depuis le cache vers les registres - ce qui est le sujet de cette sous-partie.

Load Value Prediction

On l'a vu précédemment, l'exécution spéculative est assez efficace. Du moins, elle l'est dans des cas où le résultat de la prédiction est simple. Par exemple, la prédiction de branchement est de l'exécution spéculative : on suppose que le branchement sera pris ou non-pris, et on exécute les instructions qui correspondent à ce qu'on a supposé. Les techniques de *Memory Disambiguation* qui cherchent à prédire si deux instructions accèdent à la même adresse, sont aussi de l'exécution spéculative.

Value prediction

Cette fois-ci, cela va beaucoup plus loin que prévu. Pour diminuer les effets de la *Dataflow Limit*, certains concepteurs de processeurs sont allés beaucoup plus loin. D'ordinaire, le processeur parie sur des choses simples, pour lesquelles il a peu de chances de se tromper. Un branchement est pris ou non-pris, deux adresses sont dépendantes ou ne le sont pas, etc. Dans les cas précédemment cités, on n'a que deux possibilités : pris/non-pris, dépendantes/indépendantes. De plus, on peut optimiser ces techniques de façon à utiliser certaines régularités dans nos programmes, afin de prendre de meilleures décisions, et obtenir au final de bons résultats.

Mais cette fois-ci, on change totalement de plan. Je vais vous parler des techniques de ***Value Prediction***, qui consistent à prédire quelle est la valeur présente dans un registre ou une adresse mémoire à laquelle on veut accéder. Oui, vous avez bien lus : notre processeur est capable de parier sur la valeur qui sera chargée depuis la mémoire et tenter de décider si cette valeur vaut 0, 1, 1024, etc. Une fois son pari fait, il exécute les instructions du programme avec la valeur qu'il a parié de façon spéculative. Si le pari est correct, alors on continue l'exécution. Sinon, on est obligé de faire comme lorsque l'on se trompe lors d'une prédiction de branchement ou une prédiction de dépendances d'adresses mémoires : on vide le pipeline, et on recommence avec la bonne valeur.

Au premier abord, cette technique semble franchement mal parti. Tenter de prédire quelle sera la valeur stockée dans un registre de 32 bits parmi les 4 294 967 296 valeurs que ce registre peut stocker semble être une aberration. Les chances de se tromper sont tellement énormes ! Mais le fait est que dans certains cas, il est possible de spéculer correctement. Bon, évidemment, ces cas sont plutôt rares, et dans la majorité des cas, le processeur refuse de parier. Il ne spécule pas, et n'exécute pas d'instructions en pariant sur les données qu'elle vont manipuler. Mais dans certains cas bien précis, on peut spéculer sur le résultat fourni par une instruction.

Ces cas bien précis concernent souvent le résultat fourni par une instruction de lecture en mémoire. Par exemple, on peut parier qu'une instruction de lecture qui s'exécute plusieurs fois de suite à la même adresse peut renvoyer la même valeur à chaque fois : c'est parfaitement possible si il n'y a eu aucune écriture à cette adresse entre temps. Mais ce n'est pas toujours le cas : on est donc obligé de parier. Cette technique qui consiste à parier sur le résultat d'une lecture s'appelle la ***Load Value Prediction***. Nous allons nous intéresser à cette technique dans la suite de ce tutoriel, et nous ne parlerons pas des technique qui essayent de prédire le résultats d'autres instructions (arithmétiques, etc).

Pourquoi ça marche ?

On peut se demander quelles sont les raisons qui font qu'une instruction de lecture renvoie la même valeur à chaque fois. Après tout, autant lire une seule fois la donnée et la garder dans un registre une bonne fois pour toute ! Mais cela n'est possible que dans un monde parfait. Dans la réalité, on fait face à quelques limites.

Register Spill Code

Cela notamment arriver quand on n'a pas assez de registres pour stocker toutes nos données : certaines données doivent temporairement être déplacées en mémoire pour libérer des registres, puis sont remises dans les registres une fois qu'on a des registres de libres. On peut parfaitement spéculer que lorsqu'une instruction en lecture s'exécute après une instructions d'écriture à la même adresse, la lecture renverra le résultat qui a été écrit juste avant.

Constant Pool

Cela arrive aussi quand on stocke des constantes en mémoire. Par exemple, sur les processeurs x86, les constantes flottantes ne peuvent pas être intégrées dans nos instructions via le mode d'adressage immédiat. A la place, on les stocke en mémoire et on les

charge dans les registres à chaque fois qu'on en a besoin.

Compilateurs

Il arrive aussi que les compilateurs n'arrivent pas à gérer efficacement les accès mémoires. Diverses raisons existent pour cela : certaines dépendances entre instructions forcent certaines données à être relues depuis la mémoire. Cela arrive notamment lorsque l'on utilise des pointeurs ou des références : divers phénomènes complexes d'aliasing des pointeurs peuvent générer des relectures intempestives de données en mémoire. Cela peut aussi venir de machins qui arrivent lorsqu'on compile du code qui provient de plusieurs librairies, bref.

Branchements indirects

Enfin, certains branchements indirects doivent relire l'adresse à laquelle il doive brancher depuis la mémoire régulièrement. Quand vous utilisez des switch ou des fonctions virtuelles dans votre langage objet préféré, votre compilateur va relire l'adresse à laquelle brancher (l'adresse de la fonction ou du case) depuis la mémoire à chaque accès. Il faut dire que cette adresse peut changer a tout moment, et qu'on est donc obligé d'aller la relire à chaque fois. Mais vu que cette adresse change peu, et qu'elle est souvent la même, les techniques de *Load Value Prediction* fonctionnent bien.

Implémentation

Pour implémenter cette technique, il suffit d'intégrer dans notre processeur une mémoire cache un peu spéciale. De la même façon qu'un utilise un *Branch Target Buffer* lorsqu'on parie sur les branchements, on doit utilise un cache équivalent pour la spéculation sur les lectures.

Load Value Prediction Table

Pour commencer, la première chose à faire, c'est de disposer d'un moyen pour prédire correctement si une lecture va renvoyer le même résultat que la dernière fois. Pour cela, notre processeur incorpore une unité de prédiction des valeurs lues, aussi appelée ***Load Value Prediction Table***.

Cette unité consiste simplement en une mémoire cache, sans *tag*, couplée à des compteurs. Cette mémoire cache contient l'adresse de l'instruction de lecture. Je ne parle pas de l'adresse à laquelle notre instruction va lire, mais de l'adresse à laquelle se situe l'instruction, celle contenue dans le *Program Counter* quand il exécute l'instruction de lecture. Cette mémoire cache contient donc des lignes de caches qui stockent ces adresses. Chaque adresse, chaque ligne de cache est reliée à des compteurs à saturation, similaires à ceux vus dans les unités de prédiction de branchement. Pour être franc, toutes les techniques vues dans le chapitre sur la prédiction de branchement peuvent s'adapter pour construire une *Load Value Prediction Table*. On peut aussi utiliser des compteurs à saturation simples, ou utiliser des *Two level adaptative predictor*, voire des *Agree Predictors*.

Quoiqu'il en soit, à chaque cycle, le contenu du *Program Counter* sera comparé au contenu de la mémoire cache : si l'adresse contenue dans le *Program Counter* correspond à une adresse stockée dans le cache, cela signifie qu'une instruction de lecture déjà exécutée l'est une fois de plus. Il suffit alors de regarder le résultat du ou des compteurs à saturation pour voir si on peut prédire notre instruction de lecture ou pas.

Load Value Table

Il nous faut aussi se souvenir de quelle était la valeur lue lors des dernières exécutions de notre lecture. Pour cela, rien de plus facile : on utilise une autre mémoire cache, qui contient cette valeur. Cette mémoire caches a pour *tag*, l'adresse à laquelle lire. Cette mémoire cache s'appelle la ***Load Value Table***.

Elle peut être améliorée pour conserver nos pas la dernière valeur lue, mais les n dernières valeurs lues depuis la mémoire. Cela permet de mieux s'adapter au contexte, mais est d'une utilité assez limitée en pratique. Et cela nécessite des changements dans la *Load Value Prediction Table*, qui doit prédire quelle est la bonne valeur à utiliser.

Constant Verification Unit

Et enfin, il faut bien vérifier que notre prédiction était correcte. Pour cela, rien de plus simple : il suffit de comparer la donnée fournie par la *Load Value Table*, et celle fournie par la lecture. Si elles sont identiques, la prédiction était correcte. Sinon, la prédiction était fausse, et on doit vider le pipeline. Dans tous les cas, on prévient la *Load Value Prediction Table* pour qu'elle mette à jour ses estimations et les compteurs à saturation. Cette vérification est effectuée par une unité spécialisée nommée la ***Constant Verification Unit***

Efficacité

A ce stade, je dois préciser que cette technique n'est pas encore tout à fait mature, et qu'aucun processeur ne l'implémente

encore. En tout cas, la recherche sur le sujet est encore en cours, et même si aucun processeur n'a encore implémenté de technique de ce genre, cela ne saurait tarder. Quoiqu'il en soit, l'efficacité de cette technique a déjà été étudiées grâce à des simulateurs. Suivant les études ou les programmes, on trouve des résultats qui varient pas mal. Dans certains cas, la performances baisse un peu, et dans d'autres, on peut avoir des gains de plus de 60% ! Mais dans des cas normaux, on trouve des gains de 4-5% environ. Ce qui est tout de même pas mal à l'heure actuelle.

Processeurs Multiple Issue

Dans les chapitres précédents, on a vu des processeurs qui pouvaient commencer à éxecuter une nouvelle instruction par cycle, mais pas plus. Si vous regardez bien, les processeurs, y compris les processeurs *Out Of Order*, ne peuvent commencer à éxecuter qu'une nouvelle instruction par cycle. Les techniques comme l'*Out Of Order* ne permettant que de s'approcher le plus possible de l'objectif d'une nouvelle instruction exécutée par cycle. On dit que tout ces processeurs sont des processeurs ***Single Issue***.

Ces processeurs ont un IPC (le nombre d'instructions exécutées en un seul cycle d'horloge : c'est l'inverse du CPI), qui ne peut pas dépasser 1, et qui est en pratique en dessous du fait des différents problèmes rencontrés dans notre pipeline (dépendances, instructions multi-cycles, accès mémoires, etc). Et quand on court après la performance, on en veut toujours plus : un IPC de 1, c'est pas assez ! Est-ce un problème ? On va dire que oui. Évidemment, les concepteurs de processeurs on bien trouvé une solution, une parade, bref : de quoi démarrer l'exécution de plusieurs instructions simultanément. De nouveaux processeurs sont apparus, capables d'avoir un IPC supérieur à 1 : il s'agit des processeurs ***Multiple Issue***.

Processeurs superscalaires

Dans les chapitres précédents, on a vu que nos processeurs utilisaient plusieurs unités de calcul séparées pour mieux gérer les instructions multicycles. Dans le même genre, il arrive que certains processeurs, qu'ils soient In Order ou *Out Of Order*, possèdent des unités de calcul séparées pour les nombres flottants (des FPU), ou pour gérer des calculs spéciaux. Par exemple, un processeur peut contenir une unité de calcul spécialisée dans les nombres entiers, un autre dans le calcul d'adresse, une autre pour les flottants, une autre pour les décalages, etc.

Bref, un processeur peut contenir un grand nombre d'unités de calcul, servant dans certaines situations, mais pas forcément en même temps : il arrive que certaines de ces unités soient inutilisées durant un paquet de temps. Et ces unités en plus sont très souvent inutilisées : il faut au minimum l'exécution d'une instruction multicycle pour que cela fonctionne. Si on a un gros paquet d'instructions à un seul cycle, on les exécutera les unes après les autres, sans pouvoir en lancer plusieurs indépendamment dans des unités de calcul séparées. Et la raison est très simple : sur des processeurs pareils, on ne peut lancer qu'une nouvelle instruction par cycle d'horloge, pas plusieurs.

Pour rentabiliser ces unités de calcul, divers chercheurs et ingénieurs se sont dits qu'ils serait pas mal de les remplir au maximum par des instructions indépendantes à chaque cycle d'horloge. Pour cela, ils ont inventés des processeurs qui peuvent démarrer l'exécution de plusieurs instructions simultanément, dans des unités de calculs séparées, pour les remplir au maximum. Pour que cela fonctionne, ces processeur doivent répartir les instructions sur ces différentes unités de calcul, et cela n'est pas une mince affaire. Pour cela, deux techniques existent :

- soit on répartit les instructions sur les unités de calcul à l'exécution, en faisant faire tout le travail par le processeur ;
- soit le compilateur ou le programmeur se charge de répartir ces instructions sur les unités de calcul à la création du programme.

Ces deux solutions ont leurs avantages et leurs inconvénients, et décider quelle est la meilleure des solutions est assez difficile. Il existe même une sorte de querelle entre les partisans de la première solution et ceux qui préfèrent le seconde. Mais ce fameux débat entre "brainiacs" (ceux qui veulent un processeur intelligent qui fait le café et te parallélise tout seul des instructions) et "speed demons" (ceux qui veulent que le compilateur fasse le travail à la place du processeur, parce que les circuits c'est pas gratuit) est loin d'être terminé. C'est un peu comme la guerre entre processeurs CISC et RISC, ou encore Linux versus Windows : l'informatique est pleine de ce genre de querelles stupides, rien d'étonnant à ce que le domaine de l'architecture des ordinateurs le soit aussi.

Processeurs superscalaires

La première solution est celle utilisée apr les **processeurs superscalaires**. Avec ces processeurs, la répartition des instructions sur les différentes unités de calcul se fait à l'éxecution. Ces processeurs fonctionnent comme s'ils avaient plusieurs pipelines. Le nombre de " pipelines " indiquant le nombre d'instructions différentes pouvant commencer leur exécution simultanément. Par exemple, un processeur superscalaire pouvant démarrer l'exécution de deux instructions à chaque cycle d'horloge se comportera comme s'il avait deux pipelines.

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
4				IF	ID	EX	MEM
5					IF	ID	EX
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Attention : j'ai dit qu'il se comportera comme un processeur ayant deux pipelines, pas qu'il est réellement composé de deux pipelines. Croyez-moi, cette précaution n'est pas là pour rien. Les processeurs superscalaires ne sont pas vraiment crées avec plusieurs pipelines qu'on aurait collés les uns à coté des autres : pour simplifier, tous les circuits en charge d'une étape du pipeline ne sont pas forcément dupliqués. Mais pour expliquer cela, il nous faut rentrer plus profondément dans les circuits d'un tel processeur.

Pour que notre processeur répartisse ses instructions sur plusieurs unités de calcul tout seul comme un grand (avec un peu d'aide du compilateur si besoin), il suffit juste de modifier le fonctionnement des étapes de décodage et d'Issue, et toutes les autres étapes éventuelles qu'on trouve entre le *Fetch* et les unités de calcul. C'est ainsi qu'on peut créer un processeur superscalaire : il suffit de modifier le séquenceur de façon à ce que celui-ci puisse répartir plusieurs instructions en même temps sur des unités de calcul différentes.

Superscalaire In Order versus Out Of Order

Certains processeurs superscalaire n'utilisent pas l'*Out Of Order* . Cela peut paraitre bizarre, mais c'est la vérité. Sur de tels processeurs, on peut démarrer l'exécution d'instructions consécutives simultanément si celle-ci sont indépendantes et qu'il n'y aie pas de dépendance structurelle (genre, les deux instructions doivent utiliser la même ALU, si les unités de calcul adéquates sont libres, etc). Le truc, c'est que ces processeurs sont efficaces sous certaines conditions uniquement : il faut que des instructions successives n'aient aucune dépendances. Et ce genre de situations est assez rare. Pour mieux exploiter ce genre de processeurs, on doit faire en sorte que le compilateur se débrouille pour réordonner les instructions le mieux possible. Le compilateur devient assez complexe, mais cela permet d'éviter d'avoir des circuits plus ou moins complexes dans le processeur : c'est que ça coute cher en circuits, l'*Out Of Order* !

Pour donner un exemple, on va prendre le premier processeur grand public de ce type : le fameux Pentium ! Ce processeur possède deux pipelines : un qui s'appelle le U-pipe, et l'autre qui s'appelle le V-pipe. Si deux instructions se suivent dans l'ordre du programme, le processeur est capable de les éxecuter simultanément si elles sont indépendantes : une dans chaque pipeline. On pourrait aussi citer les processeurs PowerPC, autrefois présents dans les macs, et qu'on trouve dans certaines consoles de

jeux.

Il existe des processeurs plus évolués, capables de faire mieux : les processeurs superscalaires Out Of Order. Avec ceux-ci, pas besoin que plusieurs instructions se suivent pour pouvoir être exécutées simultanément. Ces processeurs peuvent exécuter des instructions indépendantes, sous certaines conditions. Le processeur a beaucoup de travail à faire dans ces conditions. Il doit notamment vérifier quelles sont les instructions indépendantes, comme sur les processeurs superscalaires In Order. Qui plus est, il doit trouver comment répartir au mieux les instructions sur les différentes unités de calcul en modifiant l'ordre des instructions. Ce qui n'est pas simple du tout ! Mais les performances s'en ressentent : on peut exécuter un plus grand nombre d'instructions simultanées.

Fetch

Sur les processeurs superscalaires, l'unité de *Fetch* est capable de charger plusieurs instructions depuis la mémoire en même temps. Généralement, cette unité va simplement charger une instruction, ainsi que celles qui suivent. Ainsi, notre unité de *Fetch* va précharger plusieurs instructions à la fois, et le *Program Counter* est modifié en conséquence.

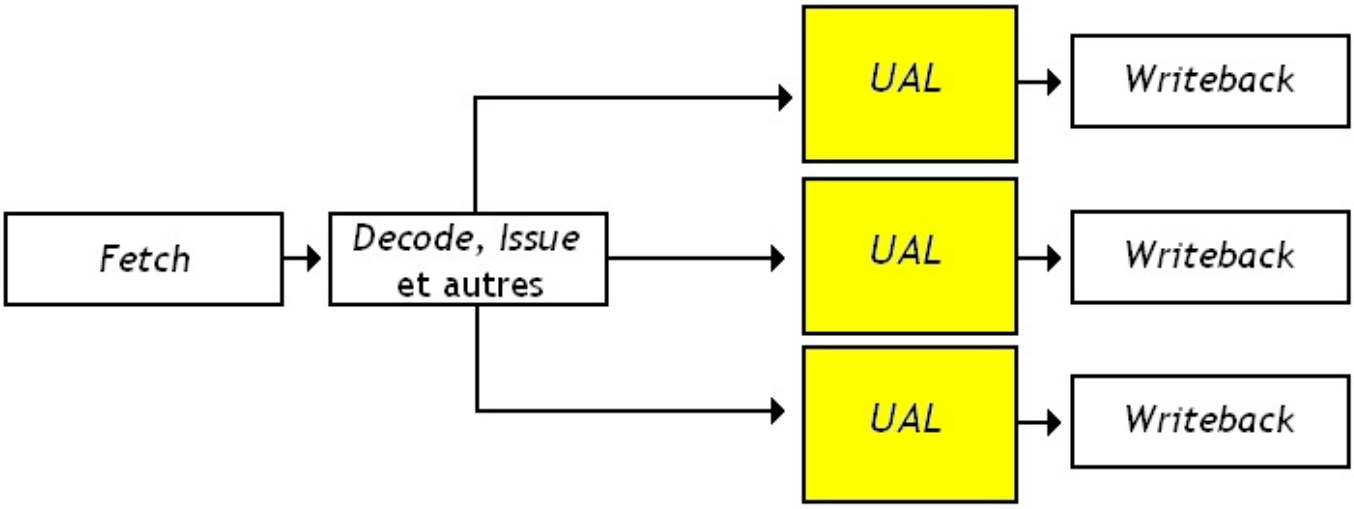
Bien sur, les branchements peuvent poser quelques problèmes dans ce genre de situations : si un branchement fait partie des instructions préchargées, que faire des instructions qui suivent : peut-on les éxecuter ou pas ? On demande alors de l'aide à l'unité de prédiction de branchement, évidemment. Mais passons cela sous le tapis.

Décodeur d'instruction

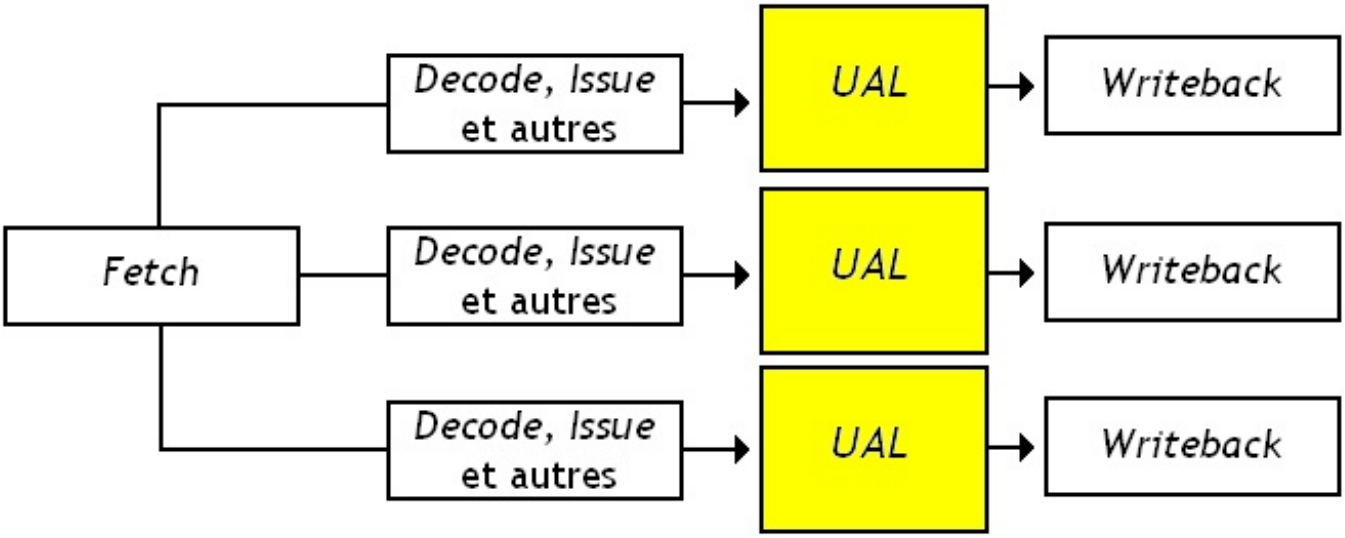
Le séquenceur est lui aussi modifié : celui-ci est maintenant capable de décoder plusieurs instructions à la fois (et il peut aussi éventuellement renommer les registres de ces instructions). Après tout, c'est normal : si on veut exécuter plusieurs instructions en même temps, il faudra bien charger et décoder plusieurs instructions simultanément !

Un ou plusieurs ?

Pour ce faire, on peut utiliser une seule unité de décodage d'instructions capable de décoder plusieurs instructions par cycle.



Mais il est aussi possible d'utiliser un processeur qui possède plusieurs séquenceurs bien séparés. On peut se débrouiller pour faire en sorte de dupliquer tous les circuits du processeur, sauf celui de *Fetch*. Évidemment, cela coute pas mal en circuits, mais cela peut être supportable.



Fusion de micro-opérations

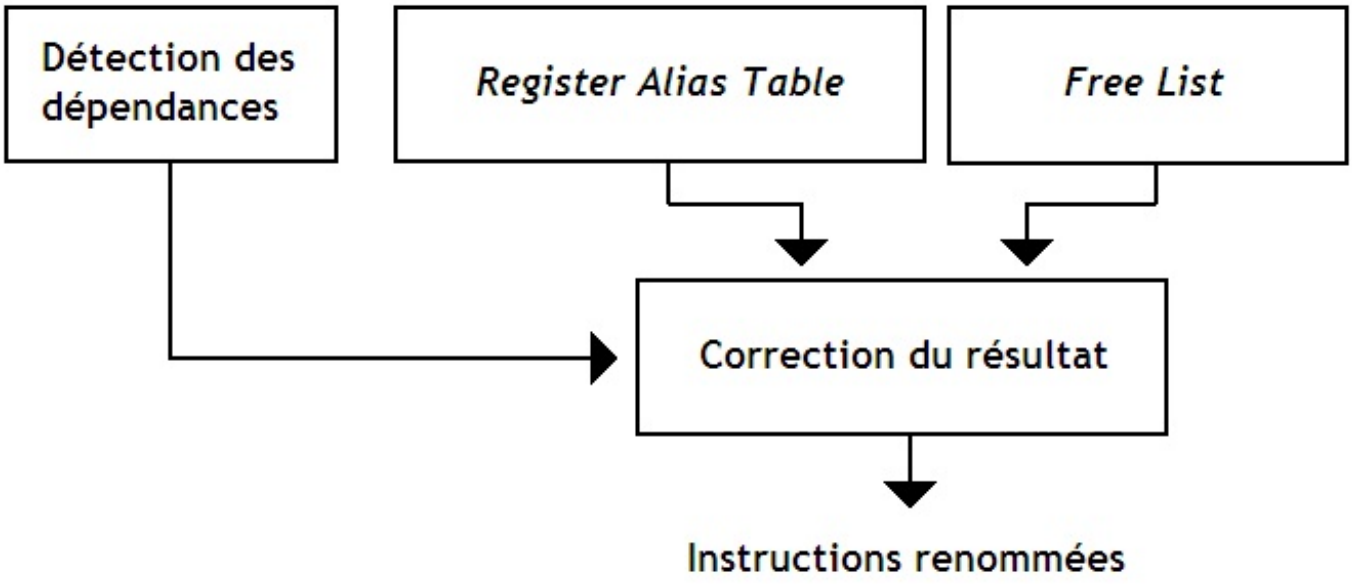
Ces processeurs superscalaires peuvent effectuer pas mal d'optimisations que d'autres processeurs ne peuvent pas faire. Dans certains cas, le séquenceur peut fusionner plusieurs instructions machines s'exécutant l'une après l'autre en une seule micro-opération. Par exemple, un processeur peut décider de fusionner une instruction de test suivie d'un branchement en une seule micro-opération effectuant le comparaison et le branchement et une seule fois. Cette dernière technique est très efficace : il faut savoir qu'environ une instruction sur 10 est un branchement précédé d'une instruction de test. Implémenter une telle fusion entre branchements et tests permet ainsi de gagner en performance assez rapidement et a pas mal d'avantages sur les architectures actuelles.

Cette fusion se fait lors du décodage de plusieurs instructions en même temps : notre décodeur d'instructions va en effet décider plusieurs instructions à la fois et peut identifier certaines suites d'instructions simplifiables.

Influence sur l'unité de renommage

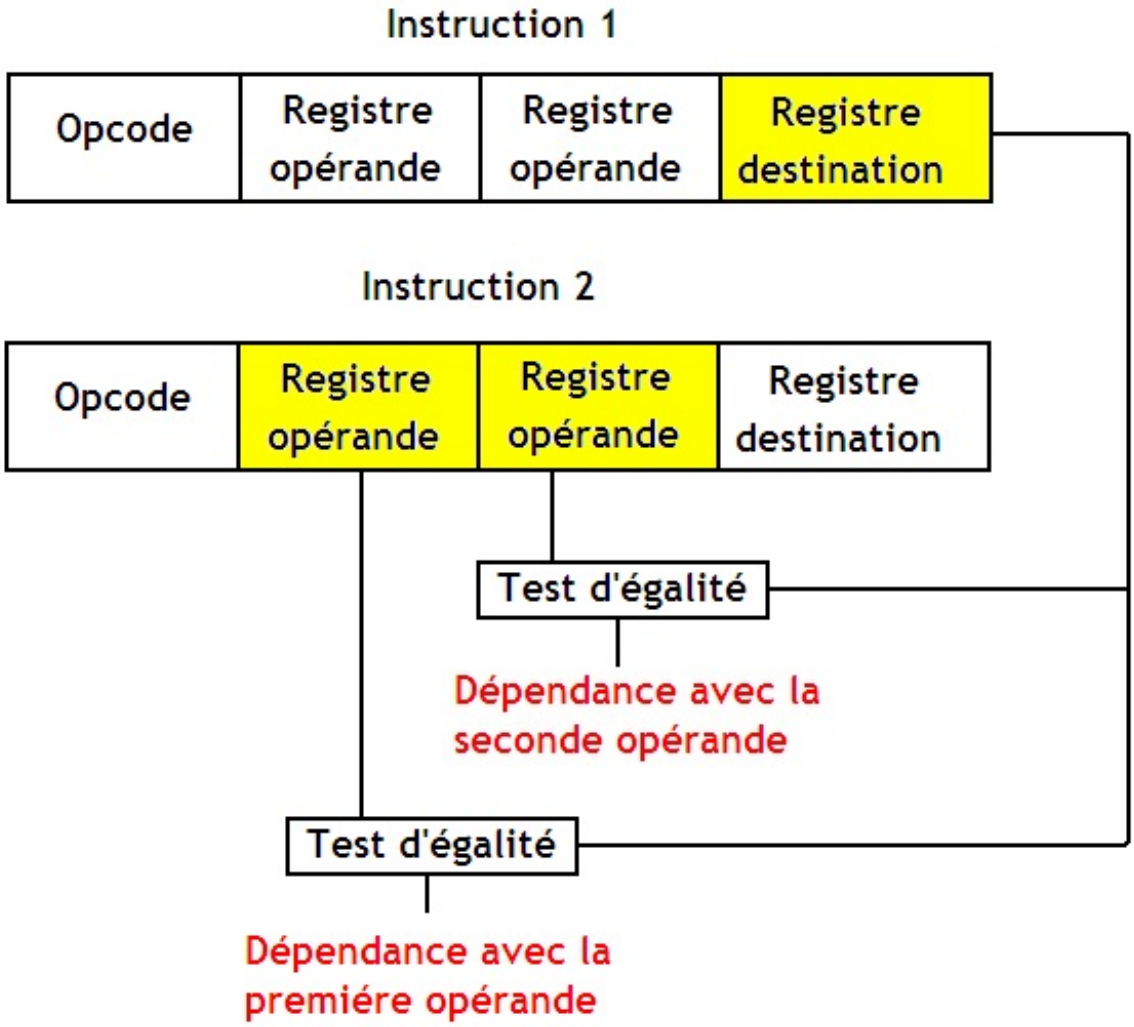
Sur les processeurs superscalaires, notre unité de renommage de registres est mise à rude épreuve. Sur les processeurs non-superscalaires, celle-ci ne devait renommer qu'une seule instruction par cycle d'horloge. Mais maintenant, elle doit renommer plusieurs instructions à la fois. Et elle doit gérer le cas où ces instructions ont des dépendances entre-elles. Cela peut se faire de différentes manières, mais on doit forcément ajouter de quoi détecter les dépendances entre instructions. Reste à savoir comment prendre en compte ces dépendances entre instructions lors du renommage de registres.

Et bien la solution est très simple : pourquoi ne pas renommer nos registres sans tenir compte des dépendances, pour ensuite corriger le résultat si jamais on trouve des dépendances ? On peut ainsi renommer nos registres et détecter les dépendances en parallèle, chaque tache se faisant dans un circuit séparé.



Détection des dépendances

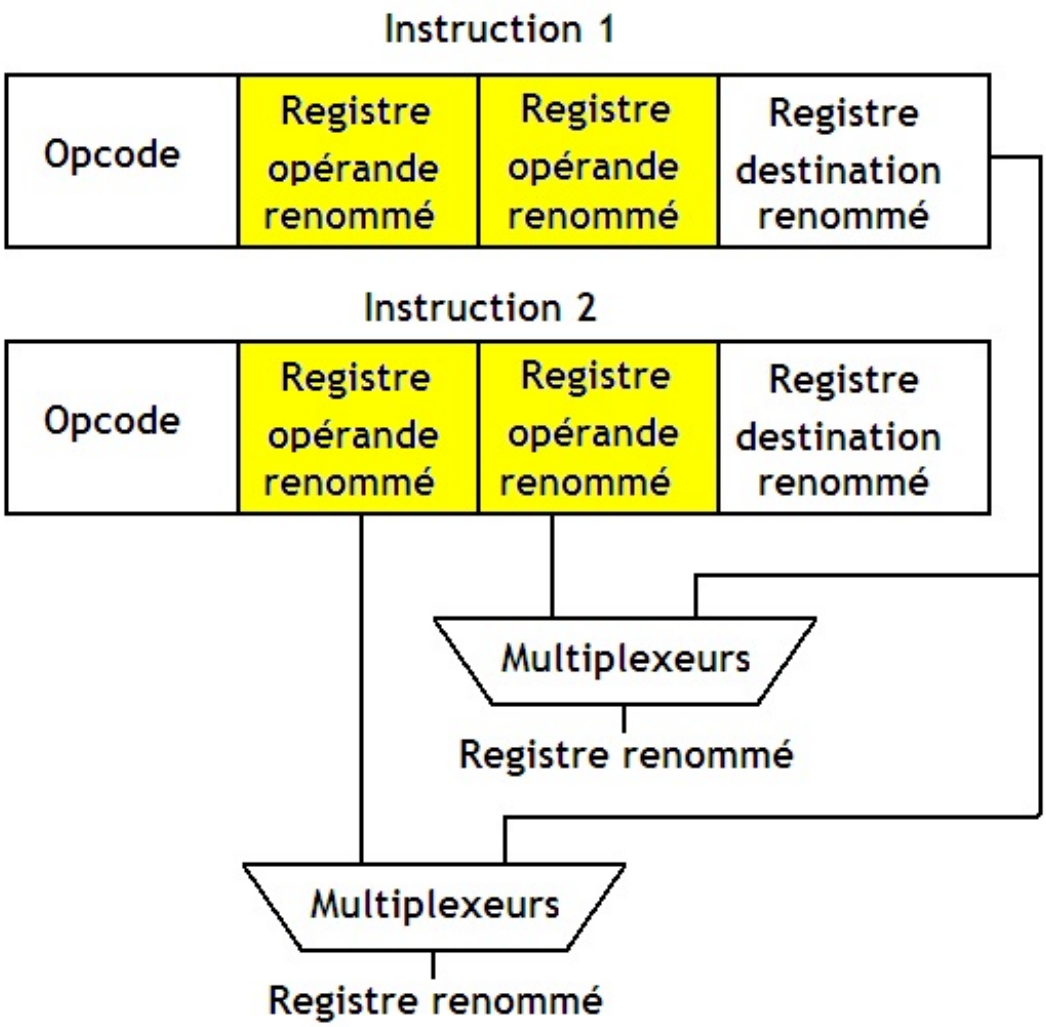
Première difficulté : comment faire pour détecter nos dépendances ? Tout d'abord, il faut remarquer que les seules dépendances que l'on doit prendre en compte sont les dépendances *Read After Write*, les autres dépendances étant justement supprimées par l'unité de renommage de registres. En clair, on doit seulement prendre en compte un seul cas : celui dans lequel une instruction a besoin du résultat d'une autre. Repérer ce genre de dépendances se fait assez simplement : il suffit de regarder si un registre de destination d'une instruction est une opérande d'une instruction suivante.



Comme vous le voyez, cela se fait simplement avec l'aide de quelques comparateurs.

Correction du résultat

Ensuite, il nous faut savoir comment corriger le résultat du renommage en fonction des dépendances. Et bien sachez que c'est super simple à mettre en œuvre : il suffit d'utiliser des multiplexeurs. Le principe est tout simple. Si une instruction n'a pas de dépendance avec une autre, on la laisse telle qu'elle. Par contre, si elle a une dépendance avec une instruction précédente, cela signifie que un de ces registre opérande sera identique avec le registre de destination d'une instruction précédente. Et dans ce cas, le registre opérande n'est pas le bon une fois renommé : on doit le remplacer par le registre de destination renommé de l'instruction avec laquelle il y a dépendance. Cela se fait simplement en utilisant un multiplexeur dont les entrées sont reliées à l'unité de détection des dépendances.



On doit faire ce replacement pour chaque registre opérande. Et vu que notre processeur superscalaire a besoin de renommer un groupe de plusieurs instructions simultanément, on doit aussi faire cela pour chaque instruction du groupe renommé (sauf la toute première, celle qui précède toutes les autres). Cela nécessite d'utiliser beaucoup de multiplexeurs.

Processeurs VLIW

Dans ce qui précède, on a vu les processeurs superscalaires. Avec ceux-ci, on prenait un processeur, et celui-ci se débrouillait pour vérifier les dépendances entre instructions et tenter de remplir les unités de calcul du processeur. Avec les processeurs superscalaires *Out Of Order* et *In Order*, surtout avec les processeurs superscalaires *Out Of Order*. Bien sûr, l'aide du compilateur est précieuse pour obtenir de bonnes performances : un bon compilateur pourra réorganiser les instructions d'un programme de façon à placer des instructions indépendantes le plus prêt possible, histoire de donner des opportunités au *Scheduler*.

Le fait est que ces processeurs superscalaires sont tout de même assez complexes, particulièrement les processeurs superscalaires *Out Of Order*. Une bonne part de leurs circuits permet d'exécuter des instructions simultanément, et ces circuits ne sont pas gratuits : ils chauffent, consomment de l'électricité, ont un certain temps de propagation qui limite la fréquence, prennent de la place, coutent "cher", etc. Certains se sont dits que quitte à faire travailler le compilateur, autant que ce soit lui qui fasse tout le boulot ! Si on déporte ce travail de réorganisation des instruction et leur répartition sur les différentes unités de calcul hors du processeur, cela fera des circuits en moins, et de la fréquence en plus. C'est ainsi que les **processeurs VLIW** sont nés.

Ces processeurs sont des processeurs *In Order* : ils exécutent les instructions dans l'ordre dans lesquelles elles sont envoyées au processeurs. Mais il y a une différence avec les processeurs superscalaires habituels : les processeurs VLIW ne vont pas regarder si deux instructions consécutives peuvent être exécutées en même temps, et ils ne vont pas non plus répartir eux-même les instructions sur les unités de calculs. Cette tache sera plus ou moins déléguée au compilateur. Pour ce faire, le compilateur va devoir garantir que les instructions qui se suivent sont strictement indépendantes. Qui plus est, il va falloir lui fournir un moyen de préciser sur quelle unité de calcul lancer l'instruction.

Bundles

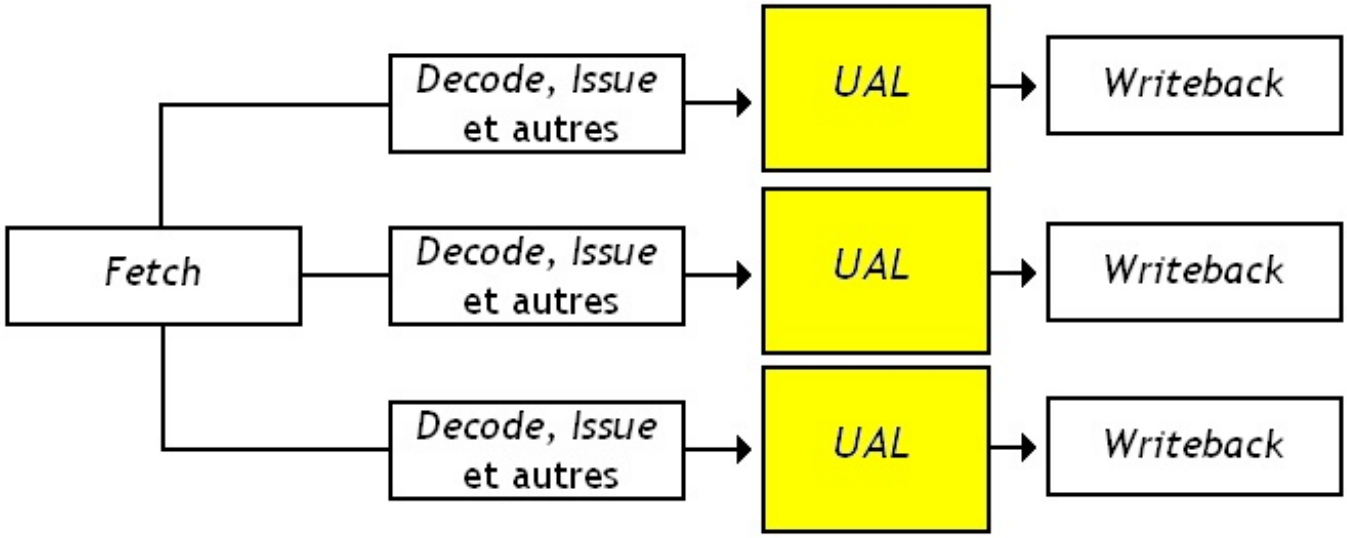
Pour ce faire, notre compilateur va regrouper des instructions dans ce qu'on appelle des *Bundles*, des sortes de super-instructions. Ces *bundles* sont découpés en *Slots*, en morceaux de taille bien précise, dans lesquels il va venir placer les instructions élémentaires à faire exécuter.

Instruction	VLIW	à 3 slots
Slot 1	Slot 2	Slot 3
Addition	Multiplication	Décalage à gauche
0111 1111 0000	0110 1111 0101	0110 1001 0101

Chaque *slot* sera attribué à une unité de calcul bien précise. Par exemple, le premier *slot* sera attribué à la première ALU, la second à une autre ALU, le troisième à la FPU, etc. Ainsi, l'unité de calcul exécutant l'instruction sera précisée via la place de l'instruction élémentaire, le *slot* dans lequel elle se trouve. Qui plus est, vu que chaque *slot* sera attribué à une unité de calcul différente, le compilateur peut se débrouiller pour que chaque instruction dans un *bundle* soit indépendante de toutes les autres instructions dans ce *bundle*.

Lorsqu'on exécute un *bundle*, il sera décomposée par le séquenceur en petites instructions élémentaires qui seront chacune attribuée à l'unité de calcul précisée par le *slot* qu'elles occupent. Pour simplifier la tache du décodage, on fait en sorte que chaque *slot* aie une taille fixe.

Exemple avec plusieurs unités de décodage.



Au passage, VLIW est un acronyme pour ***Very Long Instruction Word***. Et cet acronyme nous dit presque tout ce qu'il faut savoir sur ces architectures. Un *bundle* peut être vu comme une sorte de super-instruction très longues : ils font plus de 64 bits de long (plus de 8 octets) ! Et ils peuvent avoir une longueur pouvant aller jusqu'à 256 à 512 bits (64 octets) !

Problèmes

C'est le compilateur qui est chargé de faire tout le travail en regroupant des instructions pouvant être exécutées en même temps dans des unités de calcul différentes dans une seule grosse instruction, un seul *bundle*. On n'a donc pas besoin de rajouter des circuits électroniques chargés de répartir les instructions sur différentes unités de calculs. Mais on a intérêt à ce que le compilateur soit de qualité, parce que sinon, on est mort ! Et c'est un peu le problème de ces architectures : si le compilateur ne peut pas trouver de quoi remplir ces *bundles* avec des instructions indépendantes, il va devoir ruser. Et hormis quelques applications bien spécifiques, il est difficile de remplir ces *bundles* correctement. Pour les programmes ayant beaucoup d'instructions indépendantes, cela ne pose pas trop de problèmes : certains programmes de traitement d'image ou faisant des traitements spécifiques peuvent tirer partie des architectures VLIW. Mais ceux-ci sont loin d'être une majorité.

Sur les anciens processeurs VLIW, les instructions VLIW, les *bundles*, étaient de taille fixe. Le compilateur devait donc remplir ceux-ci totalement, et remplissait les vides avec des `nop`, des instructions qui ne font rien. Dans ce genre de cas, pas mal de bits sont utilisés dans ces bundles pour rien : ils n'encodent pas une instruction et se contentent de dire de ne rien faire à l'unité de calcul qui leur est attribué. On utilise donc de la place à rien. Le programme compilé pour une architecture VLIW sera donc plus

gros, et prendra plus de place en mémoire ou dans le cache. Ce qui peut rapidement foutre en l'air les performances si on ne peut pas remplir les *bundles* correctement. Ces processeurs doivent donc être fournis avec des caches énormes, des bus rapides, utilisent une grande quantité de mémoire, etc. Autant dire qu'en terme de consommation énergétique ou de performances, cela peut réduire à néant les performances gagnées en simplifiant le processeur.

Néanmoins, il faut citer que la majorité des processeurs VLIW récents arrive à résoudre ce problème, en utilisant les instructions de longueur variable. Ainsi, les *bundles* vides, non remplit par des instructions, ne sont pas encodés dans l'instruction. Les `nop` ne sont pas stockés dans l'instruction et ne prennent aucune place : le compilateur utilise juste ce qu'il faut pour encoder les instructions utiles, pas plus. Cela aide énormément à diminuer la *code density* des programmes compilés pour ces architectures VLIW récentes.

Et c'est sans compter que certaines dépendances entre instructions ne peuvent être supprimées qu'à l'exécution. Par exemple, le fait que les accès à la mémoire aient des durées variables (suivant que la donnée soit dans le cache ou la RAM, par exemple) joue sur les différentes dépendances. Un compilateur ne peut pas savoir combien de temps va mettre un accès mémoire, et il ne peut organiser les instructions d'un programme en conséquence. Par contre, un processeur le peu : on en touchera surement un mot au chapitre suivant. Autre exemple : les dépendances d'instructions dues aux branchements. Les branchements ont en effet tendance à limiter fortement les possibilités d'optimisation du compilateur. Alors qu'à l'exécution, un processeur peut prédire les branchements et supprimer un paquet de dépendances en fonction.

Qui plus est, ces processeurs n'ont strictement aucune compatibilité, ou alors celle-ci est très limitée. En effet, le format des super-instructions VLIW, des *bundles*, est spécifique à un processeur. Celui-ci va dire : telle instruction va sur telle ALU, et pas ailleurs. Mais si on rajoute des unités de calcul dans une nouvelle version du processeur, il faudra recompiler notre programme pour que celui-ci puisse l'utiliser, voire même simplement faire fonctionner notre programme. Dans des situations dans lesquelles on se fout de la compatibilité, cela ne pose aucun problèmes : par exemple, on utilise beaucoup les processeurs VLIW dans l'embarqué. Mais pour un ordinateur de bureau, c'est autre chose...

Processeurs EPIC

Comme on l'a vu, les architectures VLIW ont quelques problèmes. Une faible compatibilité, une *code density* pouvant être assez mauvaise. De plus, la performance de ces processeurs dépend fortement de l'efficacité du compilateur. Le but de ces architectures est simple : on délègue l'ordonnancement des instructions au compilateur, qui remplace totalement l'*Out Of Order*. En l'aidant un peu si possible. Pas d'*Out Of Order*, un peu de *Register Rennaming*, mais pas trop, et peu de techniques évoluées qui rendent le processeur capable de faire le café.

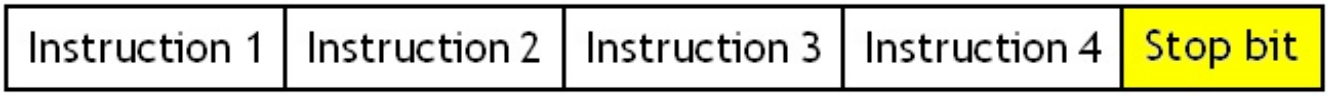
En 1997, Intel et HP lancèrent un nouveau processeur, l'Itanium. Son architecture ressemblait fortement aux processeurs VLIW, mais avec les défauts en moins. Dans un but marketing évident, Intel et HP prétendirent que l'architecture de ce processeur, bien que ressemblant aux processeurs VLIW, n'était pas du VLIW. Ils appelérent cette nouvelle architecture EPIC, pour ***Explicit Parallelism Instruction Computing***. Il faut avouer que cette architecture avait tout de même de fortes différences avec le VLIW, mais elle avait aussi beaucoup de points communs. Bien évidemment, beaucoup ne furent pas dupes, et une gigantesque controverse vit le jour : est-ce que les architectures EPIC sont des VLIW ou non ? On va passer celle-ci sous silence, et voir un peu ce que peut recouvrir ce terme : EPIC.

Bundles

Pour commencer, la première différence avec les processeurs VLIW vient de ce qu'on met dans les *Bundles*. Sur les processeurs VLIW, les instructions étaient placées dans un *Slot* bien précis qui spécifiait l'unité de calcul qui devait éxecuter l'instruction. Niveau compatibilité, c'était une catastrophe. Mais les *Bundles* des architectures EPIC ne fonctionnent pas sur ce principe.

Ceux-ci sont simplement des groupes d'instructions indépendantes. Leur place dans l'instruction ne spécifie pas l'unité de calcul qui s'occupera de l'instruction. C'est le processeur qui va découper ce *Bundles* et tenter de répartir les instructions du mieux qu'il peut sur les différentes ALU. En terme de compatibilité, c'est le rêve : on peut rajouter des ALU sans avoir besoin de recompiler. Le processeur pourra alors profiter de leur présence sans rien faire de spécial.

Ces *Bundles* sont en plus de taille variable. Avec les processeurs VLIW, les *Bundles* avaient souvent une taille fixe. Et quand le compilateur n'arrivait pas à les remplir, il laissait des vides. Ça gâchait de la RAM. Mais avec les processeurs EPIC, on n'a pas ce genre de choses. Les *Bundles* des processeurs EPIC sont délimités par un petit groupe de bits spécial, qui indique la fin d'un *Bundles*. Ce petit groupe de bits s'appelle un ***Stop Bit***.



Prédication

Enfin, on peut signaler que les processeurs EPIC possèdent un grand nombre d'instructions à prédicats. Pour rappel, ces instructions à prédicat sont des instructions "normales", comme des additions, copie d'un registre dans un autre, multiplication, accès mémoire, etc ; avec une différence : elles ne font quelque chose que si une condition est respectée, valide. Dans le cas contraire, celles-ci se comportent comme un `nop`, c'est à dire une instruction qui ne fait rien !

Utilité

Leur but ? Celles-ci servent à remplacer un grand nombre de branchements. Il faut dire que les branchements sont une véritable plaie pour les compilateurs : ceux-ci empêchent d'effectuer de nombreuses optimisations. Par exemple, ils gênent fortement la capacité du compilateur à déplacer des instructions et changer leur ordre. En supprimer le plus possible permet d'améliorer la situation. En conséquence, ces instructions à prédicats sont une véritable nécessitée sur une architecture qui ne gère pas l'*Out Of Order*.

Itanium d'Intel

Pour donner un exemple d'instructions à prédicats, je vais vous parler des instructions de l'Itanium.

L'Itanium possède plusieurs registres d'états de un bit ! Il y en a en tout 64, qui sont numérotés de 0 à 63. Chacun de ces registres peut stocker une valeur : vrai (un) ou faux (zéro). Le registre 0 est en lecture seule : il contient toujours la valeur vrai, sans qu'on puisse le modifier. Ces registres sont modifiés par des instructions de comparaison, qui peuvent placer leur résultat dans n'importe quel registre à prédicat. Elle doivent tout de même préciser le registre dans lequel stocker le résultat.

Chaque instruction à prédicat va préciser quel est le registre qui contient la valeur vrai ou faux permettant d'autoriser ou d'interdire son exécution en utilisant un mode d'adressage spécial. L'instruction s'exécutera normalement si ce registre contient la valeur vrai, et elle ne fera rien sinon. Petite remarque : une instruction peut carrément spécifier plusieurs registres. Ainsi, une instruction peut s'exécuter si deux registres à prédicats sont à vrais. Par exemple, elle peut faire un ET logique sur ces deux bits et décide de s'exécuter si jamais le résultat est true. Elle peut aussi faire un OU logique, un XOR, un NAND, etc.

Petite remarque : sur l'Itanium, presque toutes les instructions sont des instructions à prédicats. Le truc, c'est que toutes les instructions qui s'exécutent normalement, de façon inconditionnelles, sont des instructions qui vérifient le registre à prédicat r0, qui vaut vrai, et qui est lecture seule. Cela implique aussi pas mal de choses bizarres. Par exemple, des comparaisons, qui écrivent donc dans un registre à prédicat, peuvent aussi être prédicatées.

Delayed Exceptions

Les branchements ne sont pas les seules instructions qui peuvent gêner les compilateurs. Les instructions qui peuvent générer des exceptions matérielles leur pose aussi des problèmes. La raison est simple : pour qu'une exception s'exécute correctement, toutes les instructions qui précédent l'exception doivent se terminer avant qu'on effectue l'exception. Et inversement, toutes les instructions qui sont après l'exception ne doivent pas s'exécuter si celle-ci a lieu. En clair, notre compilateur ne peut pas faire passer des instructions avant ou après une de leur congénère, si cette dernière peut lever une exception. Cela limite les possibilités de réorganisation des instructions : une instruction qui peut lever une exception est considérée par le compilateur comme une sorte de barrière infranchissable. Pour résoudre le problème, l'Itanium implémente ce qu'on appelle les ***Delayed Exceptions***.

Avec la technique des *Delayed Exceptions*, le compilateur peut créer deux versions d'un même code : une qui suppose qu'aucune exception matérielle n'est levée, et une autre version qui suppose qu'une exception est levée. La première version est bien optimisée, les instructions sont réorganisée en ne tenant pas compte du fait qu'une instruction pouvant lever une exception est censée être une barrière. La seconde version est compilée normalement.

Le programme est conçu pour éxecuter la première version en premier. Une fois le code de cette version terminé, le programme va alors utiliser une instruction pour vérifier que tout s'est bien passé. S'il n'y a pas eu d'exception, alors on continue d'exécuter notre programme. Mais sinon, on branche vers la version du code non-optimisée, correcte, et on l'exécute.

Pour vérifier que tout s'est bien passé, chaque registre est associé à un bit caché, qui stocke un bit spécial. Ce bit, le bit *Not A Thing*, est mit à 1 quand le registre contient une valeur invalide. Ainsi, si une instruction lève une exception, cette exception est passée sous silence, et notre instruction continue son exécution. Elle écrira alors un résultat faux dans un registre, et le bit *Not A Thing* de ce registre est mit à 1. Les autres instructions utiliseront alors cette valeur, et auront comme résultat une valeur invalide, avec le bit *Not A Thing* mit à 1. Une fois le code fini, il suffit d'utiliser une instruction qui teste ce fameux bit *Not A Thing*, et qui branche en conséquence.

Spéculation sur les lectures

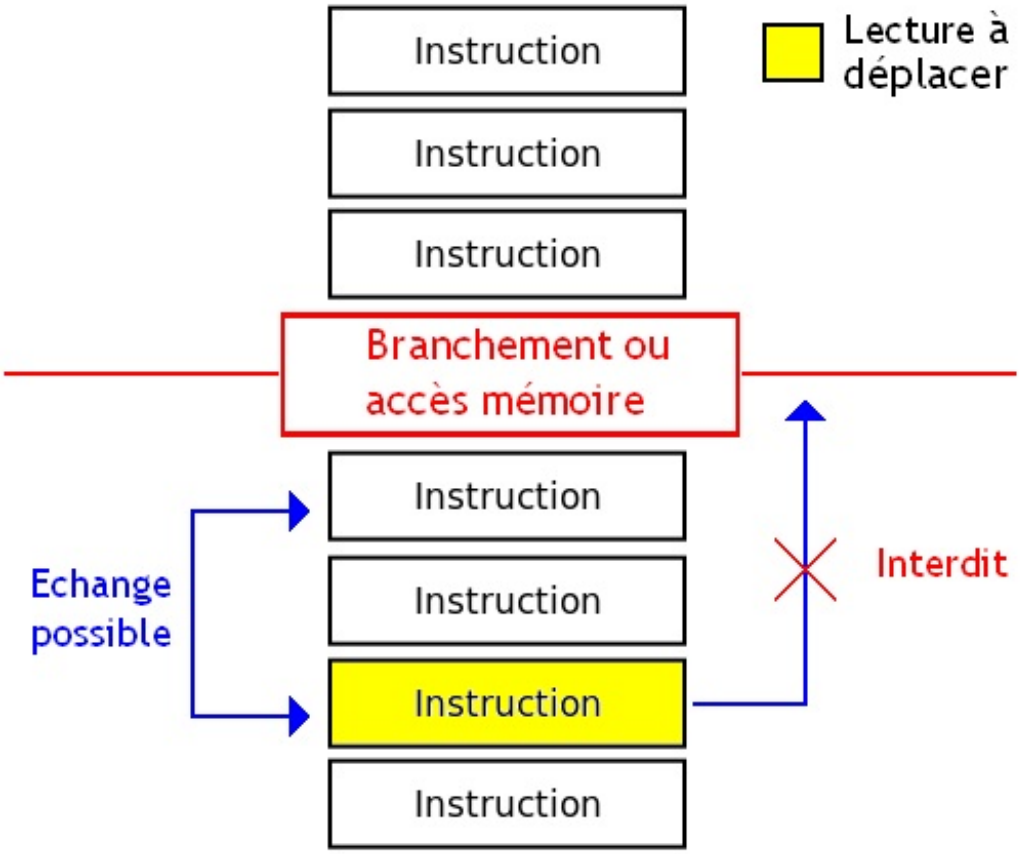
Comme on l'a vu, les *Delayed Exceptions* permettent d'aider le compilateur à réorganiser les instructions en spéculant quelque chose, et en fournissant un moyen de se rattraper en cas d'erreur. Mais l'Itanium ne s'arrête pas aux exceptions : il se charge

aussi de fournir une fonctionnalité similaire pour les instructions de lecture en mémoire.

Principe

Pour voir à quoi cela peut servir, il nous faut faire un petit rappel. Comme vous le savez, nos instructions de lecture en mémoire prennent un certain temps à s'exécuter. Cela peut aller de 2 à 3 cycles à bien plus. Pour limiter la casse, le processeur peut parfaitement éxecuter des instructions indépendantes de la lecture en attendant qu'elle se termine. Sur un processeur *Out Of Order*, le processeur et le compilateurs se chargent de d'exécuter la lecture le plus précocement possible, histoire d'exécuter un maximum d'instructions indépendantes après la lecture.

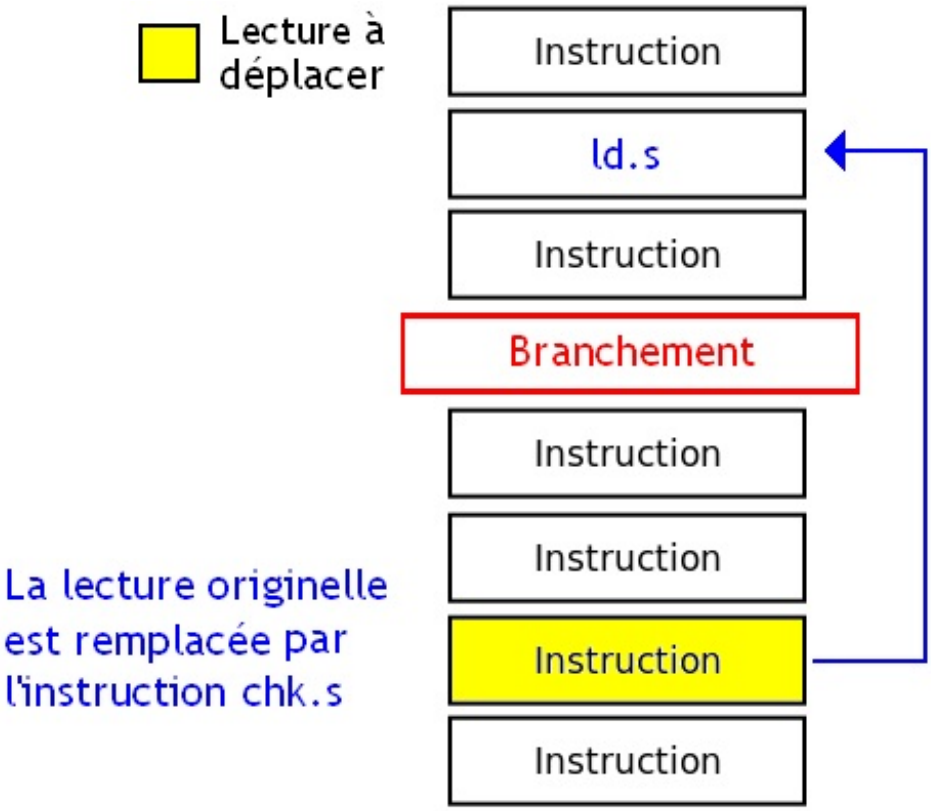
Le seul problème, c'est que sur les architectures EPIC, on est limité par le compilateur. Celui-ci ne peut pas déplacer la lecture trop tôt sans changer le comportement du programme. Par exemple, il ne peut pas déplacer la lecture avant une écriture si les adresses de ces deux instructions ne sont pas connues : si ces deux instructions travaillent sur la même adresse, ce déplacement peut changer le comportement du programme. Les dépendances RAW, WAR, et WAW vont venir jouer les trouble-fêtes. Et la même chose arrive avec les branchements : faites passer une lecture avant un branchement, et il y a un risque que ça fasse des chocapics ! Pour éviter tout problème, un compilateur a interdiction de déplacer une lecture avant une instruction de branchement, une autre lecture, ou une écriture.



Pour résoudre ce problème, l'Itanium fournit quelques instructions spéciales, qui permettent au compilateur d'exécuter des lectures en avance de façon totalement spéculative, et de revenir à la normale en cas de problème. Avec ces instructions, un compilateur peut déplacer une lecture avant une autre instruction d'accès mémoire ou un branchement. Cette lecture devient alors une lecture spéculative. Reste ensuite à vérifier que la spéculation était correcte. Si jamais la spéculation rate, une dépendance a été violée et la lecture ne renvoie pas le bon résultat. Vérifier qu'aucune dépendance n'a été violée ne se fait pas de la même façon selon que la lecture aie été déplacée avant un branchement ou avant une autre écriture.

Spéculation sur le contrôle

Si on passe une lecture avant un branchement, la lecture et la vérification sont effectuées par les instructions `ld.s` et `chk.s`.



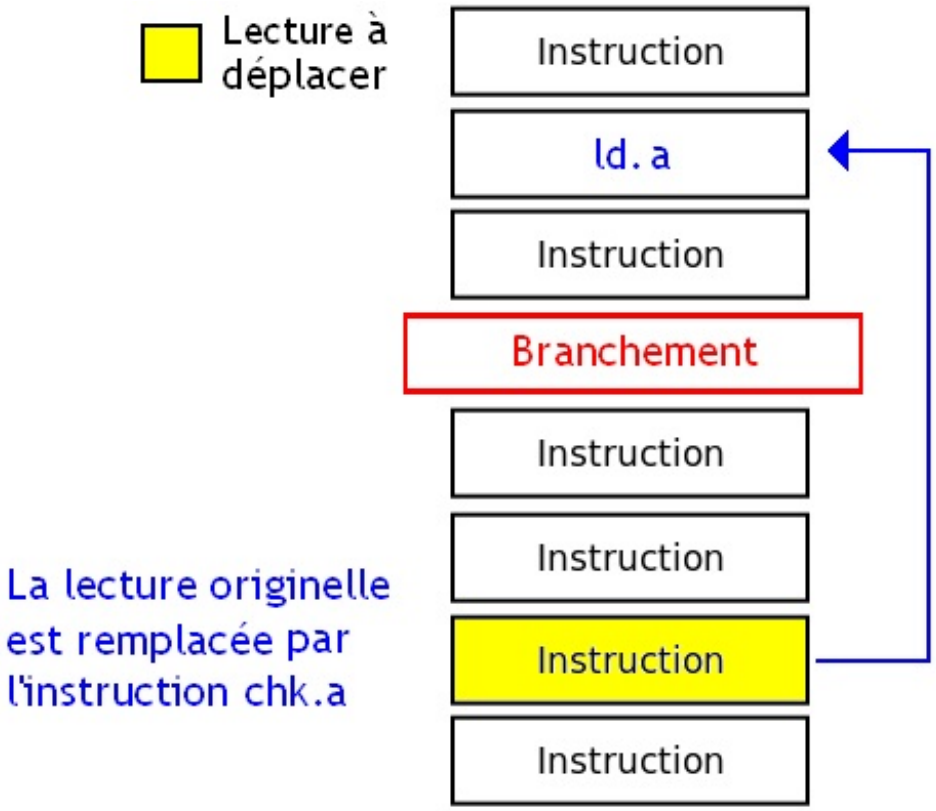
Si jamais la lecture est effectuées alors qu'elle ne devrait pas l'être, le processeur détecte automatiquement cette situation. Pour résoudre ce problème, il lève une *Delayed Exception*, et le bit *Not A Thing* du registre contenant la donnée lue est alors mit à 1. `chk.s` ne fait rien d'autre que vérifier ce bit. Si jamais la spéculation n'a pas marché, `chk.s` va brancher sur un morceau de programme qui traite cette exception lancée pas notre lecture trop anticipée.

Spéculation sur les données

Mais l'Itanium permet aussi de déplacer une lecture avant une autre écriture. D'ordinaire, un bon processeur, disposant de mécanismes de *Memory Disambiguation* pourrait le faire lui-même, et remettre le pipeline à zéro en cas d'erreur. Mais la philosophie de l'architecture EPIC est de déléguer cette tache au compilateur. Et c'est donc lui qui doit s'y coller. L'architecture EPIC fournit de quoi aider le compilateur : ce dernier peut ainsi spéculer que la lecture et l'écriture vont aller lire à des adresses mémoires différentes, et réparer l'état du processeur en cas d'erreur. Après tout, si les adresses sont dans des registres ou non connues à la compilation, le compilateur ne peut pas savoir s'il y a dépendance entre ces deux accès mémoires ou non. Spéculer dessus est donc parfois nécessaire.

La vérification des dépendances de données entre lectures anticipée et écriture se fait à l'exécution. Pour ce faire, le processeur utilise l'***Advanced Load Address Table***, aussi appelée ALAT. C'est une sorte de mémoire cache, qui stocke des informations sur nos lectures anticipées. Elle stocke notamment l'adresse qui a été lue, a longueur de la donnée lue, le registre de destination, et quelques autres informations.

Tout se passe comme avec les branchements, à part que les instructions ne sont pas les mêmes. Le processeur va devoir utiliser les instructions `ld.a` et `chk.a`.



Lorsqu'il exécute l'instruction `ld.a`, le processeur va remplir l'ALAT avec les informations sur la lecture, et va démarrer celle-ci. Puis, le processeur continue d'exécuter le programme. Si jamais une écriture à la même adresse a lieu, les informations sur la lecture sont supprimées de l'ALAT.

Pour vérifier que notre lecture s'est bien passée, on utilise l'instruction `chk.a`. Celle-ci va vérifier si tout s'est bien passé en lisant ce qui est stocké dans l'ALAT. Si il trouve une correspondance dans l'ALAT, alors la lecture s'est passée correctement. Sinon, c'est qu'une écriture l'a supprimée et que le contenu fourni apr la lecture est périmé : on doit recommencer la lecture pour obtenir un résultat correct.

Large Architectural Register File

Un processeur EPIC n'implémente aucune technique de *Register Renamming*. Ou alors s'il le fait, c'est pour des cas particuliers, et fans une forme assez bâtarde, qui concerne très peu d'instructions. Pour compenser, un processeur EPIC contient un grand nombre de registres architecturaux, afin de faciliter le travail du compilateur. Il faut dire que les compilateurs aiment beaucoup disposer d'un grand nombre de registres : cela leur permet de diminuer le nombre d'accès mémoires, en stockant un grand nombre de variables dans ces registres. Cela leur permet aussi de stocker des données dans les registres au lieu de les recalculer sans cesse par manque de place. Bref.

Ce grand nombre de registres permet au processeur d'utiliser des instructions 3-adresses. Elles permettent de préciser non seulement les registres des opérandes, mais aussi le registre de destination. Celui qui sert à stocker le résultat.

Bilan

Mettons les choses au point : l'Itanium est aujourd'hui abandonné. Il ne fut pas un franc succès. La raison : cette architecture se basait trop sur le compilateur pour gagner en performances. En théorie, il est possible d'utiliser cette architecture au maximum de ses capacité si on dispose d'un bon compilateur. Le seul problème, c'est qu'aucun compilateur n'est suffisant pour tirer correctement parti d'une telle architecture. Et c'est pas faute d'avoir essayé : l'Itanium fournissait pas mal de fonctionnalités pour aider le compilateur. Mais le problème, c'est que le compilateur ne peut pas tout, et que le *Hardware* a une meilleure capacité à optimiser un programme à la volée. Dans un code remplit de branchements, avec beaucoup de dépendances, les architectures VLIW et EPIC sont mauvaises, et les processeurs *Out Of Order* sont les rois. Cependant, la situation s'inverse souvent dans des programmes avec peu de branchements et de dépendances. Bref, la morale de l'Itanium, c'est que le compilateur ne peut pas tout, et que tenter de tout lui déléguer ne marche pas quand on cherche la Performance Ultime, avec un grand P.

Partie 8 : Annexes

Dans cette partie, nous allons voir quelques petites choses pas vraiment fondamentales, mais assez intéressantes.

Alignement mémoire et endianess

Outre le jeu d'instruction et l'architecture interne, les processeurs différent par la façon dont ils lisent et écrivent la mémoire. On pourrait croire qu'il n'y a pas grande différence entre processeurs dans la façon dont ils gèrent la mémoire. Mais ce n'est pas le cas : des différences existent qui peuvent avoir un effet assez important. Dans ce chapitre, on va parler de l'*endianess* du processeur et de son alignement mémoire : on va s'intéresser à la façon dont le processeur va repartir en mémoire les octets des données qu'il manipule. Ces deux paramètres sont surement déjà connus de ceux qui ont une expérience de la programmation assez conséquente. Les autres apprendront ce que c'est dans ce chapitre.

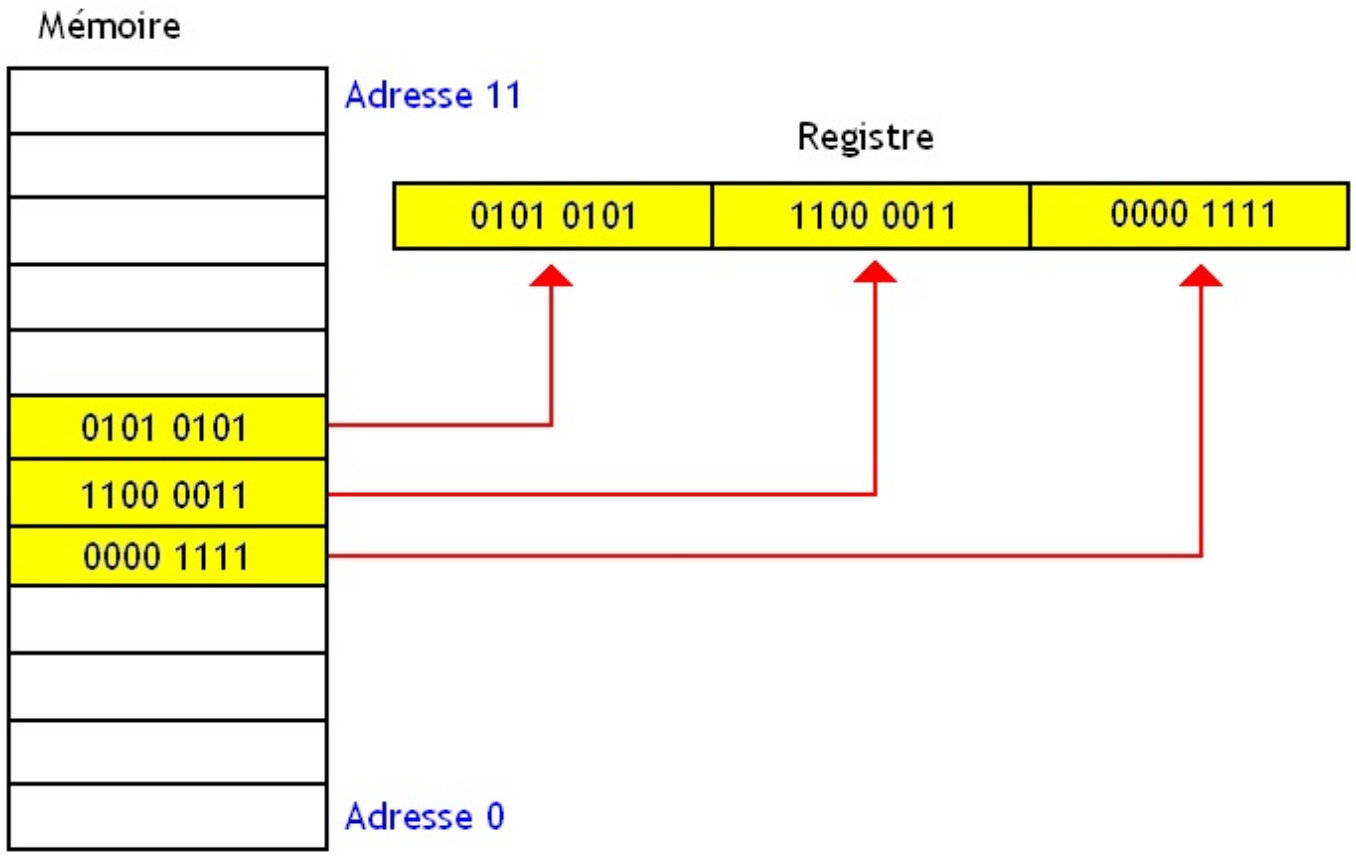
Alignement mémoire

Si vous vous souvenez des chapitres précédents, vous vous rappelez que le bus de donnée a souvent une largeur de plusieurs octets. Le processeur peut ainsi charger 2, 4 ou 8 octets d'un seul coup (parfois plus). On dit que le processeur accède un **mot** en mémoire. Ce mot n'est rien d'autre qu'une donnée qui a la même taille que le bus de donnée.

Suivant le processeur, il existe parfois des restrictions sur la place de chacun de ces mots en mémoire.

Accès mémoires à la granularité de l'octet

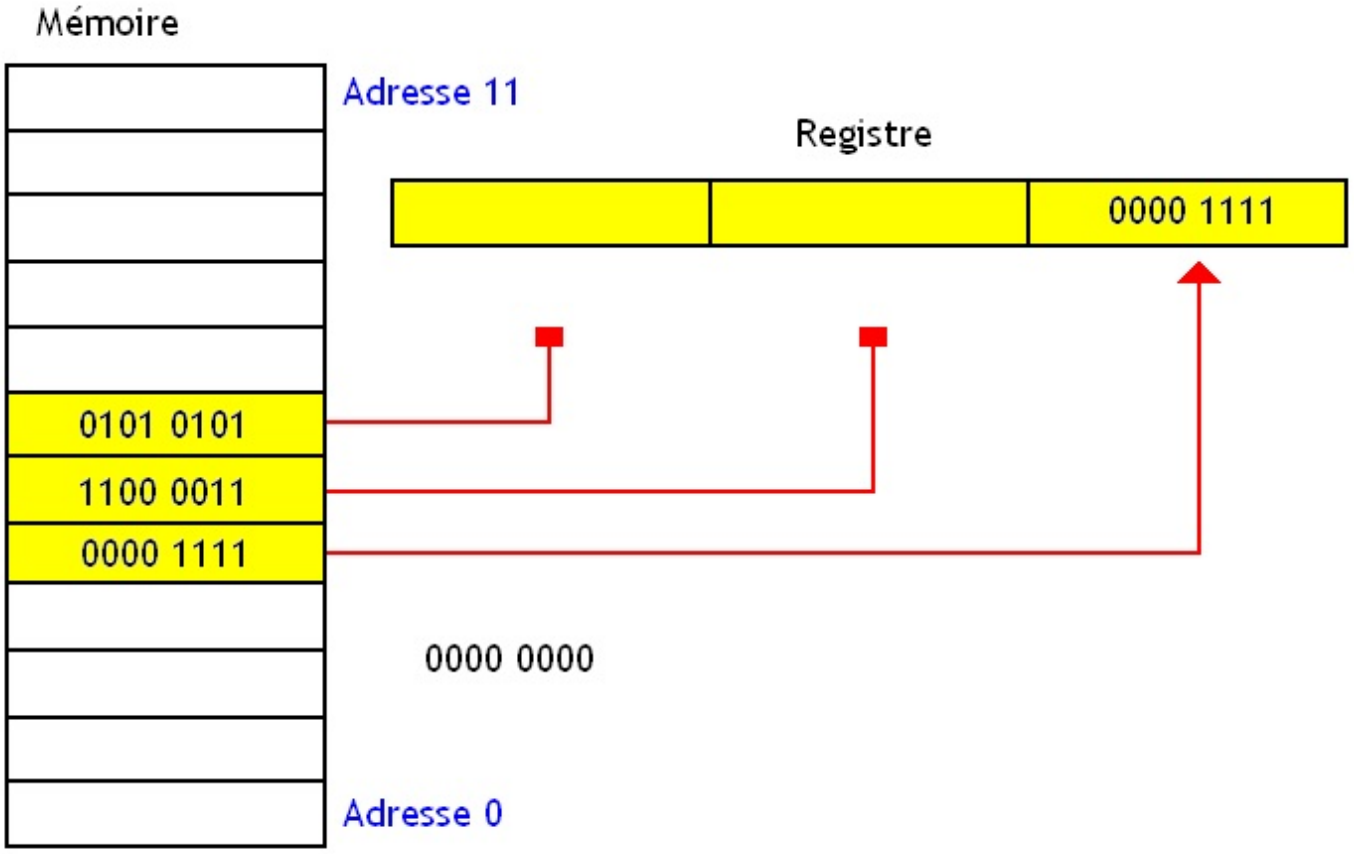
On peut voir la mémoire comme un tableau de cases mémoires, faisant chacune un octet, toutes accessibles individuellement.



On peut parfaitement vouloir lire 1, 2, 4, 8 octets individuellement. Par exemple, on peut vouloir charger un octet depuis la mémoire dans un registre. Pour cela, notre processeur dispose de modes d'adressages, voir d'instruction différentes, suivant qu'on veuille lire ou écrire 1, 2, 4, 8, 16 octets.

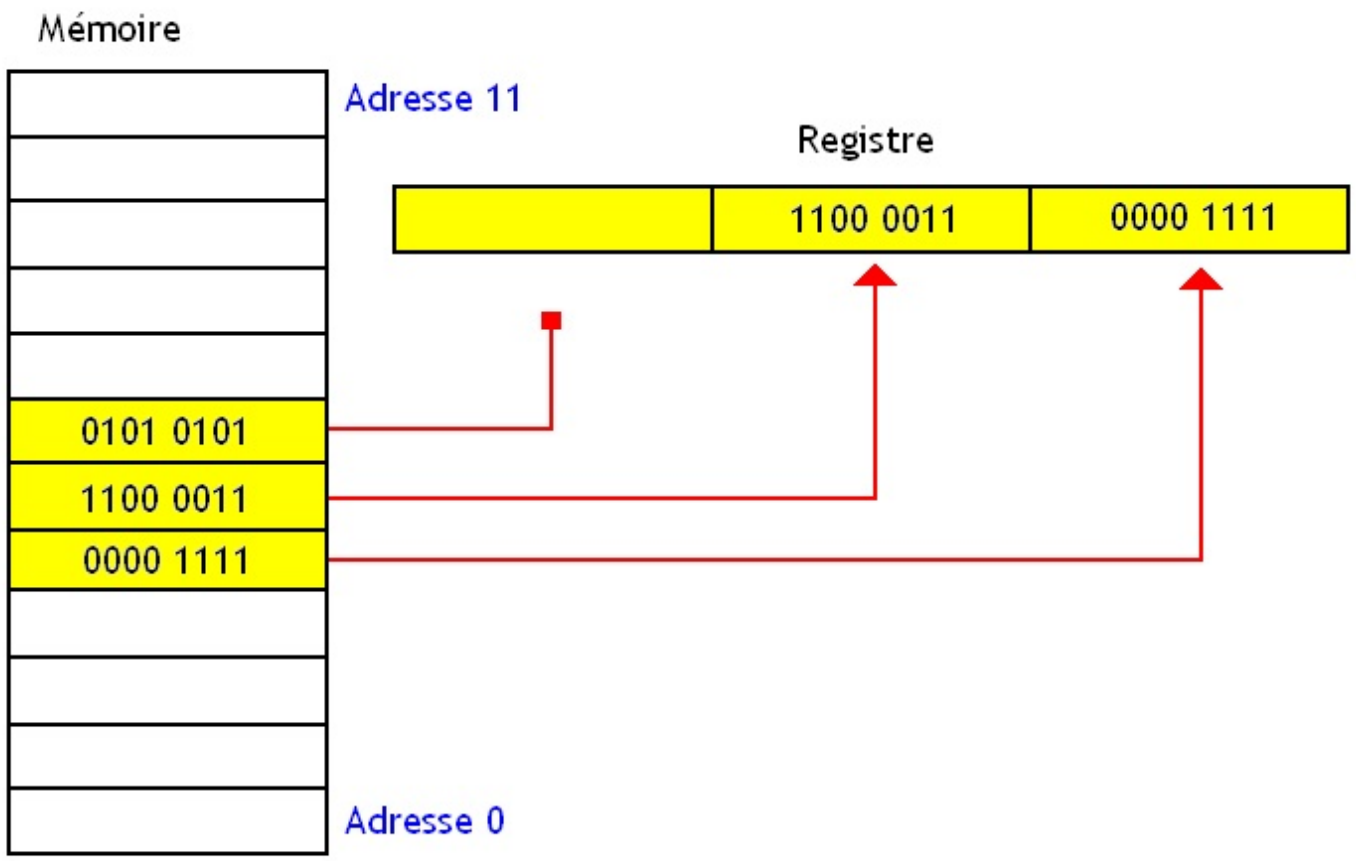
Quand on veut charger une donnée sur un bus plus grand que celle-ci, les cases mémoires immédiatement suivantes sont aussi copiées sur le bus. Mais rassurez-vous : le processeur les ignore. En choisissant la bonne instruction ou le bon mode d'adressage, les bits en trop chargés sur le bus de donnée ne seront pas pris en compte. Bien sûr, cela marche aussi pour l'écriture.

Exemple : on souhaite charger un octet dans un registre de trois octets, en passant par un bus de trois octets.

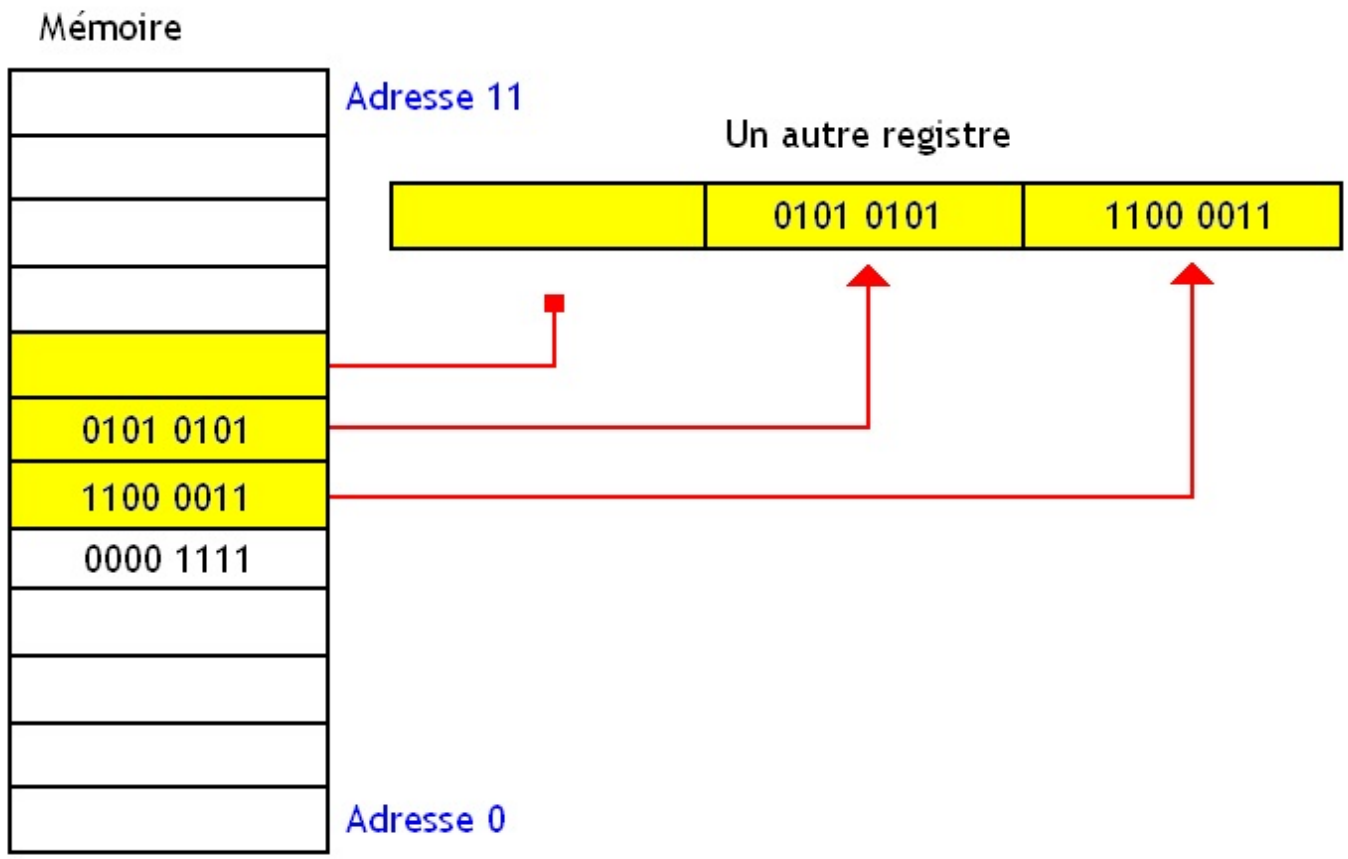


Sur de tels processeurs et mémoires, on peut lire ou écrire à n'importe quelle adresse, sans aucune restriction. Toute donnée est accessible en une seule fois, du moment que celle-ci est plus petite que le bus de donnée : elle peut faire 1, 2, 4, 8 octets, si le bus peut contenir celle-ci, on peut la charger en une seule fois quelque soit la situation, quelque soit son adresse. On dit que ces processeurs accèdent à la mémoire à la granularité de l'octet.

Pour donner un exemple, je peux parfaitement décider de charger dans mon registre une donnée de 16 bits localisée à l'adresse 4, puis lire une autre donnée de 16 bits localisée à l'adresse 5 pour la charger dans un autre registre sans aucun problème.

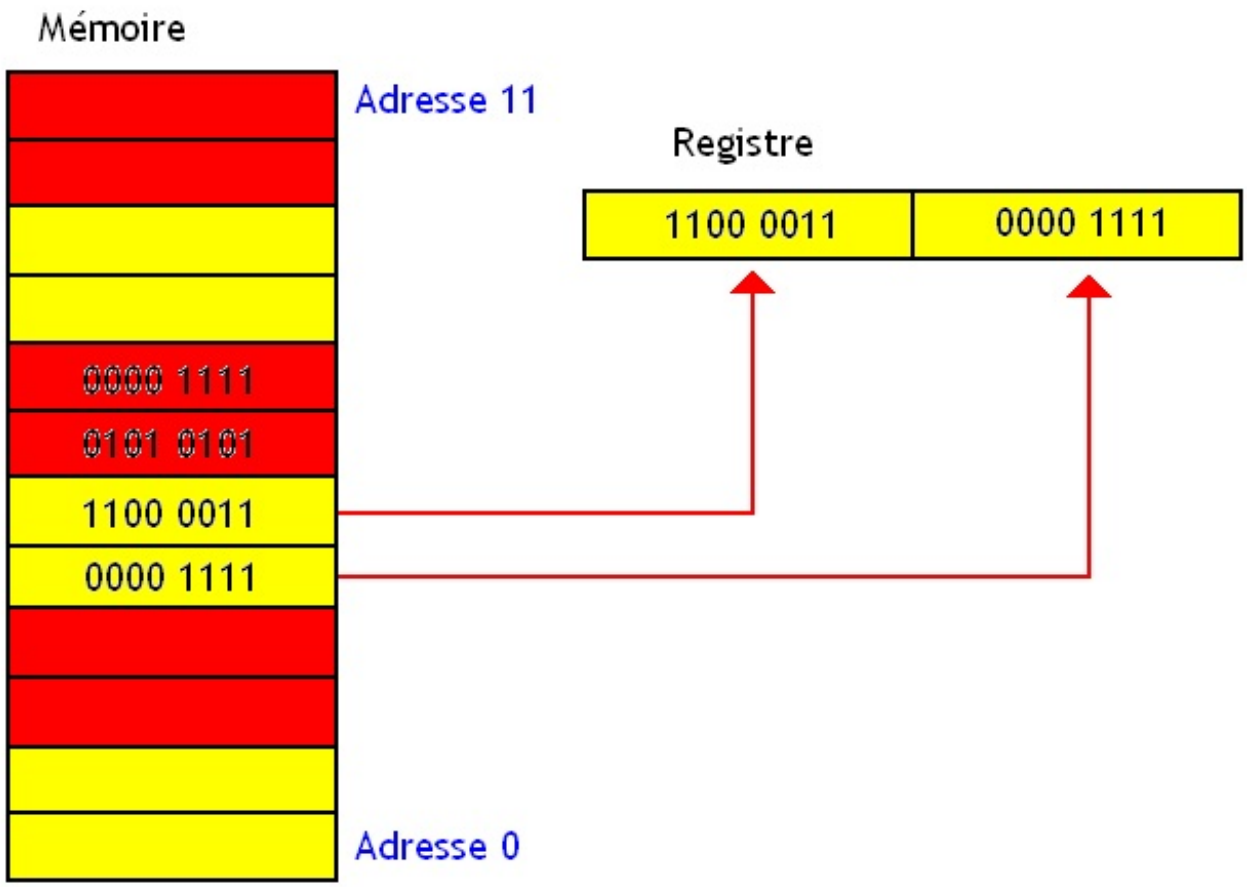


Un peu plus tard...



Alignement supérieur à l'octet

Mais d'autres processeurs ou certaines mémoires imposent des restrictions assez drastiques dans la façon de gérer ces mots. Certains processeurs (ou certaines mémoires) regroupent les cases mémoires en "blocs" de la taille d'un mot : ceux-ci utilise un certain **alignement mémoire**. On peut voir chacun de ces blocs comme une "case mémoire" fictive un peu plus grosse que les cases mémoires réelles et considérer que chacun de ces blocs possède une adresse. L'adresse d'un de ces groupes est l'adresse de l'octet de poids faible. Les adresses des octets situé dans le groupe (c'est à dire autre que l'octet de poids faible) sont inutilisables : on ne peut adresser qu'un groupe, via son octet de poids faible, et charger l'intégralité de ce mot sur le bus, mais pas accéder à un octet en particulier.



L'adressage de la mémoire est donc moins "fin" : on travaille sur des blocs de plusieurs bits, plutôt que sur des petits paquets.

Bus d'adresse et alignement

On l'a vu, certaines adresses mémoires deviennent inutilisables : celle qui sont situées dans un mot et qui ne pointent pas vers son octet de poids faible. Par exemple, si on prend un groupe de deux octets, on est certain qu'une case sur deux sera inutile : les adresses impairs pointeront sur l'octet de poids fort de notre groupe.

Prenons un autre exemple : un processeur ayant des mots de 4 octets. Répertorions les adresses utilisables : on sait que l'adresse zéro, est l'adresse de l'octet de poids faible de notre premier mot. L'adresse 1 est située dans notre mot, pareil pour 2, pareil pour 3. L'adresse 4 est utilisable : c'est l'adresse du premier octet du second mot, etc.

Voici ce que vous allez obtenir au final :

Adresses utilisables
0
4
8
12
16

20
...

Ceux qui sont observateurs remarqueront que seules les adresses multiples de 4 sont utilisables. Et ceux qui sont encore plus observateurs remarqueront que 4 est la taille d'un mot. Dans notre exemple, les adresses utilisables sont multiples de la taille d'un mot. Et bien sachez que cela fonctionne aussi avec d'autres tailles de mot que 4. En fait, ça fonctionne même tout le temps !

Si m est la taille d'un mot, alors seules les adresses multiples de m seront utilisables.

Dans la réalité, ces blocs ont une taille égale à une puissance de deux : cela permet de faire quelques bidouilles sur le bus d'adresse pour économiser des fils. Si la taille d'un mot est égale à 2^n, seules les adresses multiples de 2^n seront utilisables. Hors, ces adresses se reconnaissent facilement : leurs n bits de poids faibles valent zéro. On n'a donc pas besoin de câbler les fils correspondant à ces bits de poids faible et on peut se contenter de les connecter à la masse (le zéro volt vu dans le second chapitre).

Accès mémoires non-alignés

Bon, maintenant imaginons un cas particulier : je dispose d'un processeur utilisant des mots de 4 octets. Je dispose aussi d'un programme qui doit manipuler un caractère stocké sur 1 octet, un entier de 4 octets, et une donnée de 2 octets. Mais un problème se pose : le programme qui manipule ces données a été programmé par quelqu'un qui n'était pas au courant de ces histoire d'alignement, et il a répartit mes données un peu n'importe comment.

Supposons que cet entier soit stocké à une adresse non-multiple de 4. Par exemple :

Adresse	Octet 4	Octet 3	Octet 2	Octet 1
0x 0000 0000	Caractère	Entier	Entier	Entier
0x 0000 0004	Entier	Donnée	Donnée	-
0x 0000 0008	-	-	-	-

Pour charger mon caractère dans un registre, pas de problèmes : celui-ci tient dans un mot. Il me suffit alors de charger mon mot dans un registre en utilisant une instruction de mon processeur qui charge un octet.

Pour ma donnée de 2 octets, pas de problèmes non plus ! Mais c'est parce que mon processeur est prévu pour. Dans ce genre de cas, il suffit que je donne à mon instruction l'adresse à laquelle commence ma donnée : ici, ce serait l'adresse 0x 0000 0009. Je ne donne pas l'adresse du mot, mais l'adresse réelle de ma donnée. L'instruction détectera que ma donnée est stockée intégralement dans un mot, chargera celui-ci, et fera en sorte de n'écrire que la donnée voulue dans mon registre.

Mais si je demande à mon processeur de charger mon entier, ça ne passe pas ! Mon entier est en effet stocké sur deux mots différents, et on ne peut le charger en une seule fois : mon entier n'est pas *aligné en mémoire*. Dans ce cas, il peut se passer des tas de choses suivant le processeur qu'on utilise. Sur certains processeurs, la donnée est chargée en deux fois : c'est légèrement plus lent que la charger en une seule fois, mais ça passe. Mais sur d'autres processeurs, la situation devient nettement plus grave : notre processeur ne peut en effet gérer ce genre d'accès mémoire dans ses circuits et considère qu'il est face à une erreur, similaire à une division par zéro ou quelque chose dans le genre. Il va alors interrompre le programme en cours d'exécution et exécuter un petit sous-programme qui gérera cette erreur. On dit que notre processeur effectue une **exception matérielle**.

Si on est chanceux, ce programme de gestion d'erreur chargera cette donnée en deux fois : ça prend beaucoup de temps. Mais sur d'autres processeurs, le programme responsable de cet accès mémoire en dehors des clous se fait sauvagement planter. Par exemple, essayez de manipuler une donnée qui n'est pas "alignée" dans un mot de 16 octets avec une instruction SSE, vous aurez droit à un joli petit crash ! C'est pas pour rien que ce genre d'instructions est si peu utilisé par nos compilateurs.

Pour éviter ce genre de choses, les compilateurs utilisés pour des langages de haut niveau préfèrent rajouter des données inutiles (on dit aussi du *padding*) de façon à ce que chaque donnée soit bien alignée sur le bon nombre d'octets. En reprenant notre exemple du dessus, et en notant le *padding* X, on obtiendrait ceci :

Adresse	Octet 4	Octet 3	Octet 2	Octet 1
0x 0000 0000	Caractère	X	X	X

0x 0000 0004	Entier	Entier	Entier	Entier
0x 0000 0008	Donnée	Donnée	X	X

Comme vous le voyez, ça prend un peu plus de place, et de la mémoire est gâchée inutilement. C'est pas grand chose, mais quand on sait que de la mémoire cache est gâchée ainsi, ça peut jouer un peu sur les performances. Il y a aussi des situations dans lesquelles rajouter du *padding* est une bonne chose et permet des gains en performances assez abominables : une sombre histoire de cache dans les architectures multiprocesseurs ou multicores, mais je n'en dit pas plus. Moralité : programmeurs, faites gaffe à bien gérer l'alignement en mémoire !

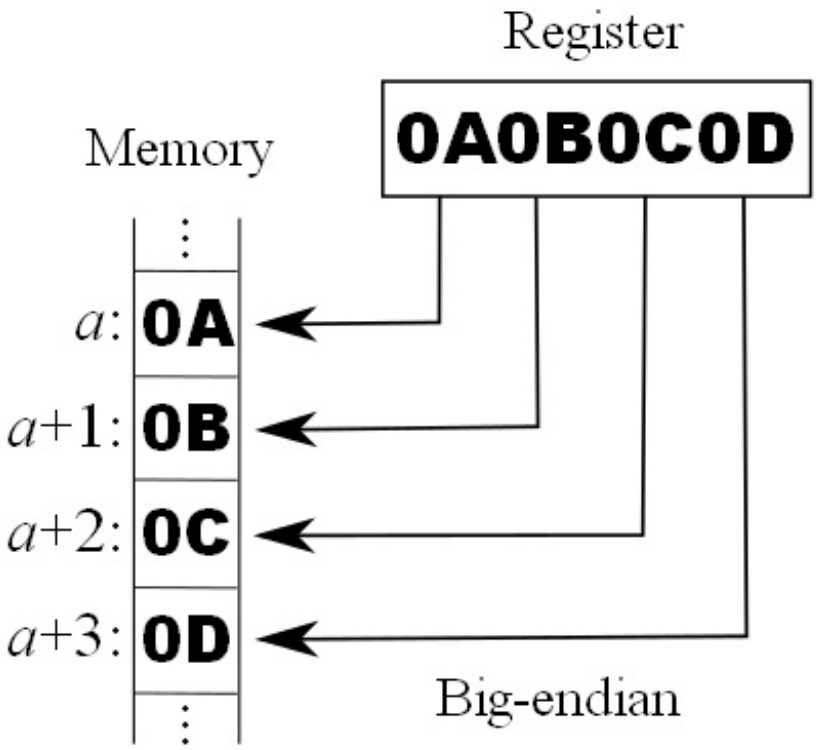
Cet alignement se gère dans certains langages (comme le C, le C++ ou l'ADA), en gérant l'ordre de déclaration de vos variables. Essayez toujours de déclarer vos variables de façon à remplir un mot intégralement ou le plus possible. Renseignez-vous sur le *padding*, et essayez de savoir quelle est la taille de vos données en regardant la norme de vos langages.

Endianness

Vous savez tous, depuis le premier chapitre, comment représenter des données simples (des nombres entiers, flottants) en binaire sous la forme d'une suite de bits. Vous savez que votre donnée sera représentée sous la forme d'une suite de bits stockée dans des octets qui se suivent dans la mémoire. Mais il y a une chose que vous ne savez pas encore : la façon dont ces bits seront repartis dans des octets varie suivant le processeur !
On peut faire une analogie avec les langues humaines : certaines s'écrivent de gauche à droite (le français, l'anglais) ; et d'autres de droite à gauche. Dans un ordinateur, c'est un peu pareil : nos données ont diverses sens d'écriture. Lorsqu'ils écrivent des données prenant plusieurs octets en mémoire, ceux-ci vont les écrire de gauche à droite dans un octet, ou l'inverse. Quand on veut parler de cet ordre d'écriture, on parle d'***endianness***. Il existe divers *endianness*, variables suivant le processeur. Voyons lesquels et ce qui peut les différencier.

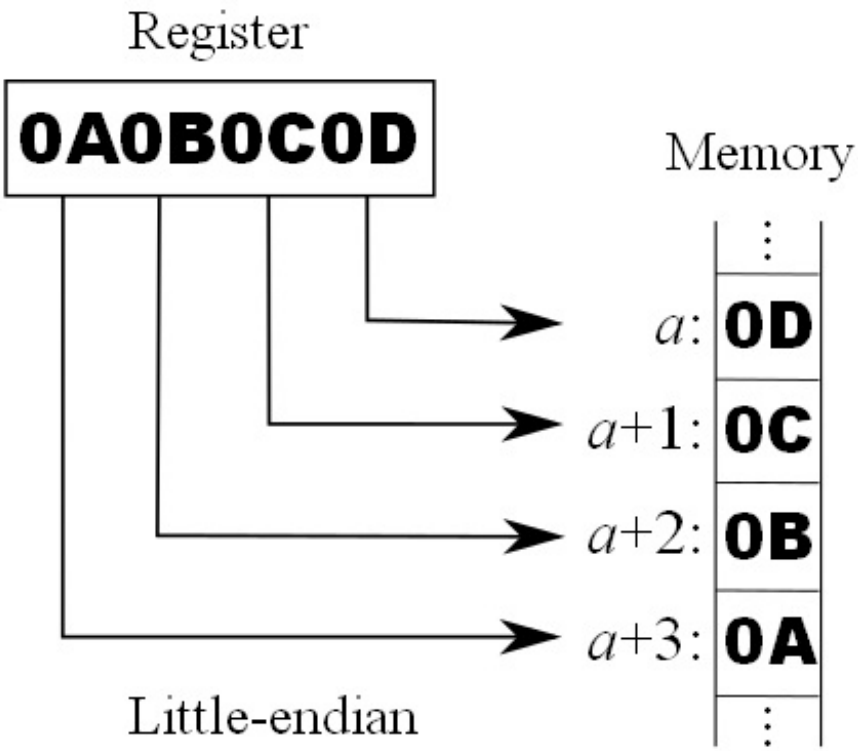
Big Endian

Certains processeurs sont de type *Big endian*. Sur ces processeurs, l'octet de poids fort de notre donnée est stocké dans la case mémoire ayant l'adresse la plus faible.



Little Endian

Sur les processeurs *Little endian*, c'est l'inverse : l'octet de poids faible de notre donnée est stocké dans la case mémoire ayant l'adresse la plus faible.



Un des avantages de ces architectures est qu'on peut accéder à une donnée quelques soit sa longueur sans changer l'adresse de la donnée à charger, même avec les contraintes dues à l'alignement. En *Big endian*, il faudrait changer l'adresse à laquelle accéder en fonction du nombre de *bytes* à lire ou écrire.

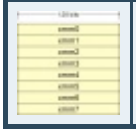
Bi-Endian

Certains processeurs sont un peu plus souples : ils laissent le choix de l'*endianness*, et permettent de le configurer de façon logicielle ou matérielle. Ces processeurs sont dits *Bi-endian*.

Liens sur le Siteduzéro

Pour poursuivre, je me permets de vous présenter trois tutoriels sur les processeurs, présents sur le siteduzéro, tous complémentaires à ce qui a été écrit dans ce tutoriel. **N'hésitez pas à lire ces tutoriels** !

Icône	Titre
	Multicoeurs, Hyperthreading, parallélisme : qu'est-ce que c'est ?
	Technologies de gestion de la consommation électrique et du TDP d'un processeur
	Les processeurs *Dataflow* : parallélisme et langages fonctionnels
	Le BIOS : qu'est-ce que c'est ?
	Les mémoires associatives

	Les Caches adressés par somme (Sum Adressed Caches)
	Les processeurs orientés objet
	Nombres flottants et processeurs

Remerciements

Un petit merci à :

- à Spacefox pour son travail de validateur attitré du tutoriel,
- Macros le noir pour sa relecture attentive de mon tutoriel,
- et à Lucas-84, pour l'ensemble de son œuvre.